Individual Service Provisioning

Individual Service Provisioning

BY

FREDRIK ESPINOZA

A Dissertation submitted to Stockholm University in partial fulfilment of the requirements for the Degree of Doctor of Philosophy

Department of Computer and Systems Sciences
Stockholm University and
Royal Institute of Technology

December 2002

Stockholm University and
The Royal Institute of Technology
Dept. of Computer and Systems Sciences
Forum 100
164 40 Kista
Sweden

Swedish Institute of Computer Science
P. O. Box 1263
S-164 29 Kista
Sweden

DSV Report series No. 03-002
ISSN 1101-8526
ISRN SU-KTH/DSV/R–03/2–SE

SICS Dissertation Series 31
ISSN 1101-1335
ISRN SICS-D–31–SE

Doctoral Thesis
Stockholm University
ISBN: 91-7265-556-9

Copyright by Fredrik Espinoza 2002
Typeset by the author using LaTeX
Printed in Sweden by Akademitryck AB, Edsbruk, 2002

Abstract

Computer usage is once again going through changes. Leaving behind the experiences of mainframes with terminal access and personal computers with graphical user interfaces, we are now headed for handheld devices and ubiquitous computing; we are facing the prospect of interacting with *electronic services*. These network-enabled functional components provide benefit to users regardless of their whereabouts, access method, or access device. The market place is also changing, from suppliers of monolithic off-the-shelf applications, to open source and collaboratively developed specialized services. It is within this new arena of computing that we describe *Individual Service Provisioning*, a design and implementation that enables end users to create and provision their own services. Individual Service Provisioning consists of three components: a personal service environment, in which users can access and manage their services; ServiceDesigner, a tool with which to create new services; and the provisioning system, which turns end users into service providers.

Acknowledgments

I wish to thank my supervisor Carl Gustaf Jansson; my advisers Kristina Höök, Magnus Boman, and Annika Wærn; my former and present coworkers Per Persson, Petra Fagerberg, Mark Tierney, Andreas Espinoza, Olle Olsson, Anna Sandin, Stina Nylander, Peter Lönnqvist, Mikael Boman, Hanna Nyström, Nils Dahlbäck, Anna Jonsson, Anette Hulth, and Terje Lundin; and the *core business* sView inner group: Markus Bylund, Lucas Hinz, and last and foremost, Ola Hamfors. Fgdbdd.

I also wish to thank the rest of the members of the HUMLE laboratory at SICS, as well as a few other people in the SICS staff: Janusz Launberg for his optimism and support, Eva Gudmundsson and Charlotta Jörsäter for much patience, Björn Gambäck for help with LaTeX, and Mikael Nehlsen for hard-core tech support.

Finally, thank you Nick, Iz, Andy (again), and Camilla.

Fredrik Espinoza
December 2002

List of papers

This thesis is composed of the following papers. In the summary, they will be referred to as papers A through E.

A. Fredrik Espinoza. *sicsDAIS: Managing User Interaction with Multiple Agents*. Licentiate of Philosophy Thesis, Stockholm University, 1998.

B. Markus Bylund and Fredrik Espinoza. sView – Personalized Service Interaction. In Jeffrey Bradshaw and Geoff Arnold, editors, *Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 2000)*, pages 215–218, Manchester, UK, April 2000. The Practical Application Company Ltd.

C. Fredrik Espinoza and Ola Hamfors. ServiceDesigner: A Tool to Help End-Users Become Individual Service Providers. In *Proceedings of HICSS-36, Hawaii International Conference on System Sciences*, January 2003.

D. Fredrik Espinoza and Lucas Hinz. Generic Peer-to-Peer Support for a Personal Service Platform. In *Proceedings of The 2003 International Symposium on Applications and the Internet (Saint 2003)*, January 2003.

E. Fredrik Espinoza. Towards Individual Service Provisioning. Short paper in *Proceedings of Intelligent User Interfaces (IUI 2003)*, January 2003.

Papers are reprinted with permission of the respective publisher:

Paper C: ©2003 IEEE. Reprinted, with permission, from Proceedings of HICSS-36, Hawaii International Conference on System Sciences.

Paper D: ©2003 IEEE. Reprinted, with permission, from Proceedings of The 2003 International Symposium on Applications and the Internet.

Other Relevant Papers

These papers, although not included, also constitute part of my thesis work:

1. Patricia Charlton, Fredrik Espinoza, Ebrahim Mamdani, Olle Olsson, Jeremy Pitt, Fergal Somers, and Annika Wærn. Using an Asset Model for Integration of Agents and Multimedia to Provide an Open Service Architecture. In *Proceedings of ECMAST97: Second European Conference on Multimedia Applications*, pages 635–650. Springer-Verlag, May 1997.

2. Patricia Charlton, Fredrik Espinoza, Ebrahim Mamdani, Olle Olsson, Jeremy Pitt, and Fergal Somers. An Open Agent Architecture Supporting Multimedia Services on Public Information Kiosks. In *Proceedings of PAAM 97: Practical Applications of Intelligent Agents & Multi-agent Systems*, pages 445–466, 1997.

3. Fredrik Espinoza. sicsDAIS: A Multi-Agent Interaction System for the Internet. In *Proceedings of WebNet 99–World Conference on the WWW and Internet*, pages 1257–1258, 1999.

4. Olle Olsson and Fredrik Espinoza. Multimedia Dialogue Management in Agent-Based Open Service Environments. In H. Leopold and N. Garca, editors, *Proceedings of Multimedia Applications, Services and Techniques - ECMAST'99 4th European Conference (LNCS 1629)*, pages 515–533. Springer-Verlag, 1999.

5. Magnus Boman, Markus Bylund, Fredrik Espinoza, Mats Danielson, and David Lybäck. Trading Agents for Roaming Users. In *Proceedings of the Tokyo Mobile Roundtable*, Tokyo, May 2002. CD-rom.

Table of Contents

List of Figures

Chapter 1

Introduction

In the near future, *service centric networks*, such as the World Wide Web, broadband networks, and home networks, in combination with new devices such as portable hand-held computers and mobile phones, will enable and require a major change in the use of computer systems. The basic unit of function is the *electronic service* and *Individual Service Provisioning*,[1] the subject of this thesis, aims to create an environment in which users and service providers can coexist and thrive as services are created and distributed with the overall purpose of tending to users' needs.

The Web, starting out as an interconnected network of servers enabling the interchange of text documents, has grown to encompass a wide range of content types, including images, audio, and moving pictures. Currently, many of the Web's offerings are based not on static content but rather on dynamically assembled and user-personalized services. Users can purchase goods, search for information, control and interact with appliances, and communicate; the limits to what is possible seem to be governed only by the imagination of the service providers. Electronic services in the more specialized networks are usually tailored for use in each specific network, i.e., services in the home network provide interactive access to appliances or functions in the home, and services in the enterprise network provide access to Enterprise Resource Planning systems, such as finance, human resources, manufacturing, and warehouse systems.

In many cases, services are accessed using proprietary methods that fit the context of the usage environment. For example, services in the home area network can be accessed using a general interface on the television set, a screen-fridge device such as Electrolux's Screen Fridge (Ele, 1999), or E2-home's home appliance control system (E2 , 2002); services in mobile phone

[1] *Provisioning*: the act of supplying with a stock of needed materials or supplies (from *Merriam-Webster's Collegiate Dictionary*).

operators networks can be accessed using specially built access methods for the particular network (Sweden's largest phone operator Telia allows their customers to access voice mail using a DTMF-based interface over the customers phone). In other cases, services are made available to users by means of open and standardized solutions such as web-based interfaces or Wireless Application Protocol (WAP[2]) and Short Message Service (SMS) for mobile phones. Personal Digital Assistants (PDAs) present yet another opportunity for service interaction. These devices, along with powerful and capable mobile phones, now rival earlier personal and workstation computers in terms of performance and memory capacity. With wireless network connectivity, available through wireless local area networks using open protocols such as 802.11 and the next generation of mobile networks (3G), such devices can also access remote information and services.

In this thesis we will repeatedly consider the *service*. As the meaning of this term may not be completely well-defined in itself it is appropriate that we choose a definition. For the purpose of the discussion it is also useful to differentiate the service from the *application*. The following definitions will be used throughout this thesis.

> An ***application*** is a set of functions and abilities, packaged as a unit. It is typically installed on a target device and used locally, with its main functionality not requiring a network connection to some external component or back-end. You pay for it once, if at all, to use freely thereafter. You might not actually *own* the application; instead you may be licensed to use it (the license may also dictate any number of parameters of usage, such as transferring the application to a third party, and rules for copying the application for backup purposes). The most characteristic attribute of an application is its installation on a device.
>
> A ***service*** is a set of functions and abilities, packaged as a unit and *manifested* locally on a device—instead of being installed on the device, it is made available when needed. Its main functionality may even reside elsewhere; in which case the service uses a network to access this back-end. Another difference concerns the philosophy of charging for usage: payment is based on the amount of usage of the service—you pay for using it, not for owning it. There are several possible billing schemes, e.g., per use, per time unit, and flat rate. The most characteristic attribute of a service is its on-demand availability.

[2]WAP is a protocol used for micro browsing in mobile phones.

The following scenarios, featuring the example user, Michael, further illustrate the service concept.

Scenario 1. Michael is traveling by train to see his parents. On the day of his trip, he walks to the train station, and as he approaches the station, he looks at his watch to see the display: "Train *123* to *Malmö* @ *12:50*: track *10*, *on time*". He also glimpses the current time—12:40—and decides to go straight to track 10 instead of passing through the station house. From the electronic train service, Michael has learned that the train is leaving from track 10 and that it is on time. Michael's service briefcase system has activated, placed, and prioritized the service on the watch display a certain period of time before the train is scheduled to depart. This saves Michael the trouble of going into the station house to gather this information.

Scenario 2. Michael is visiting his parents. As the family dinner grows lengthy, Michael realizes that he will not get the chance to watch his favorite TV-show (using his parents' primitive television amenities). He stealthily extracts his mobile phone from his pocket and connects to his personal service briefcase. In the briefcase, he keeps a proxy service to the video recorder in his own home, and with a few clicks of the phone buttons, he has programmed it to record. He pockets the phone and continues conversing.

Scenario 3. On the train home from his parents', Michael decides to watch a movie on the seat-back entertainment system. The system is completely automated to minimize involvement from the busy train personnel, and as Michael logs in, it interfaces with his service briefcase to prepare for the impending payment transaction. In his briefcase, Michael has pre-configured his bank service to handle all requested payment transactions. After browsing through the available selection of movies, Michael chooses a movie and is presented with—and accepts—the familiar and trusted payment confirmation dialog of his bank service, when the entertainment system requests payment.

1.1 Hypothesis

The ultimate goal of Individual Service Provisioning is to achieve a user platform in an electronic service world, in which the right electronic services are quickly available at the right times, and in which information about users' experiences and needs can be collected and leveraged by the overall user

community. With the appropriate tools, users will be able to tailor services to their specific needs; in some cases the tailoring will go so far as to become construction—the assembly of smaller services that work together to perform a more complicated task. Thus, it becomes possible for individual users to create their own services—and with the underlying Internet infrastructure—to share them with others, which is the essence of Individual Service Provisioning.

Hypothesis 1.1.1 *With tools to* USE*,* CREATE*, and* PROVIDE *electronic services, specifically sView for using, ServiceDesigner for creating, and Briefcase Connectivity and SharedServices-Loader for provisioning services, it will be possible for end users to single handedly create and provide their own specialized services with a small effort and at a relatively low cost.*

1.2 Method

This section describes the method used for the present research. First, it explains within which domain the research has been undertaken, second, it explains the concrete methods which have been used, and, finally, it qualifies and constrains the work.

1.2.1 Domain of Research

This thesis describes a design and an implementation of a novel system for human interaction with a new breed of computing systems. It may be characterized as belonging in the cross section of Human Computer Interaction (HCI) and Software Engineering (SE), and more specifically in the area of Ubiquitous Computing and service centric computing. This position poses something of a dilemma in terms of choice of research methodology, as neither of the methods traditionally used within the fields of HCI and SE is completely applicable here.

In Human Computer Interaction research, computing system are built and tested for performance in relation to their human users. Typically, in HCI, a research effort starts by examining the demands and constraints of the human operators in the applicable domain. This can be done, for example, by experimental quantitative measurements, using ethnographic methods, surveys, or interviews. The acquired knowledge is then used in combination with applicable theories gathered from sources such as HCI, psychology, group dynamics, or any other pertinent field, to form a baseline for designing

and building a system which will try to address a number of clearly identified hypotheses. The finished system is then tested in respect to the issues, by studying its usage in laboratory experiments or in real world situations. The findings from the user studies can be used as feedback for redesigning the system or for making judgments about the identified hypotheses in relation to the proposed solutions.

Software engineering research has a somewhat different composition. First requirements are gathered. The requirements are of course very much tied to the domain, and consequently, the process of gathering the requirements also depends on the domain. For example, when creating a software system which will be used by human users, the requirements gathering resembles the examination of users in HCI: users may be observed, questioned, etc. For building an algorithm based system, the requirements may be gathered from the usage scenario, i.e., the interfacing components or related systems. Using the requirements, the system may be designed and implemented. Finally, during the testing stage, the system is evaluated relative to the requirements, and the test results give feedback into the development loop which may be performed iteratively until the requirements are met.

The present work attempts to assume a holistic user focus regarding Individual Service Provisioning with a grounding not only in HCI or SE, but rather in a combination of the two. Although the work has a bearing on HCI on an encompassing level, and many of the contributions have a significant technical grounding in SE, the main effort of the work has been focused on the user's situation, not on specific domains such as interface design, operating systems, distribution algorithms, or formal declarative languages. This choice of focus necessarily implies a (relatively) shallow coverage of certain such technical topics and it also suggests alternative research methods for the overall work. As we do not consider full scale testing of the individual parts to be the main focus of the work, the main effort has been placed in view of the total system. Parts of the work, however, do benefit from the traditional methods, and in those cases such methods have been used in a limited scope (see below).

The present work is thus an amalgamation of two areas of research and the contributions derive from this cross section examination.

1.2.2 Methods Used

The research method used for the present work is characterized by visionary and experimental development of working prototypes and surrounding infrastructure, testing of the experimental components by our colleagues and ourselves, and incremental and iterative improvement based on experiences

(cf. Weiser, 1993). The development is enclosed within a framework of literature study, design, and algorithmic and user-centered testing.

The main parts of this work, the sView personal service environment, the ServiceDesigner, the Briefcase Connectivity peer-to-peer[3] system, and the service-sharing framework, build on the visions of the sicsDAIS prototype system (Sect. 3.2.1). The four parts have been developed iteratively using the experimental and iterative approach with requirements analysis, software design and implementation, and testing. Performance tests (scalability and general performance) were conducted on the sView system and the Briefcase Connectivity system. A two-step user study was performed on the ServiceDesigner (Sect. 3.3.3).

Other tests and studies could have been performed on the specific software components; they could have been tested in laboratory settings regarding performance and compared to other similar systems (in those cases where such exist). All of the components also have some type of interface toward humans (although these interfaces are not critical to their operation). These interfaces could have been tested as to their usability in full user studies in laboratory settings. We chose to refrain from doing further component specific testing, however, since it is the complete system which is at issue here and component specific findings would be of limited value. It would have been difficult to do any type of real world testing, however, since that would have required a significant deployment effort to several systems and users. Consequently, and as a result of the main focus of the work being the creation of a working prototype system for individual service provisioning, we decided to concentrate on suggesting the feasibility of the overall system.

Finally, one issue remains to be examined; the question as to the degree in which Individual Service Provisioning will actually be found useful and be embraced by users. Our view is that there are two possible approaches to answer this question: either by performing qualitative interviews with potential users, in which the system is described and its usefulness is determined, or by deploying and trying the system in a real setting with real users, wherein its usefulness can be determined by logging of usage data. The first approach was deemed to be less valuable since the novelty of the system would in all likelihood taint the results. The second method, although it is more promising, would require a great number of users and a long time period to perform. Our standpoint here, however, is that the issue is not sufficiently relevant to warrant the expense of such a study—naturally, users' appreciation of the

[3]Peer-to-peer based systems provide a service based on a more or less decentralized architecture, with contributions from many nodes in a network. Peer-to-peer is discussed more extensively in Sect. 3.4.

usefulness of a system is important, but a system's deemed usefulness may very well grow with time and user maturity (c.f. the Web at its conception contra now).

1.2.3 Proviso

This thesis suggests a general framework that aims at making electronic services *abundant* and *ubiquitous*. The basic premise is that the need for a service originates with a user, either from personal experience or from the experiences of the user's environment. Consider mobile phone service: as a first time user starts to make and receive phone calls, he may at some point feel the need for a method to perceive (and perhaps acknowledge) calls that come in when he is indisposed and cannot answer the phone. This may be a vague need or a more precise request depending on the user's experience (for example, with an ordinary landline answering machine). Alternatively, the user may get a suggestion from an acquaintance that such a service exists and for what purpose it may be used. If the user is aware of how to go about procuring a service, and the cost to procure a service is less than the perceived benefit of the service, he will do so.

Some users can and will take matters into their own hands and create a fitting service. In the open source development community, for example, this is the way that most of the work actually gets done (Yamauchi, Yokozawa, Shinohara and Ishida, 2000). This kind of entrepreneurial effort should be encouraged and supported, by incentive building schemes, or by technical infrastructure (as shown in Sect. 3.1.1, on service provisioning). Then, if the overall service environment supports it, the gain of one user can benefit the whole user community.

There are, however, some possible issues with this view of what constitutes a service; different users may have considerably different conceptions of what services are—some may not even acknowledge a particular concept as a service. For example, when such a user performs a basic withdrawal transaction at an Automated Teller Machine (ATM), he may not consider this using a service; for this user, the whole bank concept may be the service.

Bearing this disclaimer in mind, this thesis assumes the first standpoint: some users, who conceptualize services as described in the definition above, can and will create their own services, and other users can benefit from this.

1.3 Motivation

For some time, major software vendors have indicated a need to move from the application model of using computer software, toward the service model. With services, they argue, deployment, usage, billing, and maintenance, are simplified. With a mature Internet, such a scheme is now possible.

Other large companies, in the infrastructure segment, have proposed and successfully deployed systems that support the service scheme. Sun Microsystems' Java programming language and related software make it possible to create portable and dynamically downloadable code, which is often a prerequisite for the service scheme. Furthermore, Sun also describes ways of deploying and distributing the Java-based services (Sun Microsystems, Inc., 2001). The emerging service model is discussed in Sect. 2.

Industry has also proposed the set of protocols and tools that constitute *Web Services*, network accessible functional components that may be called upon to perform functions in a truly reusable fashion. The purpose of each of these efforts, for example, Microsoft's *.NET*, Sun's *Open Network Environment*, and IBM's *Web Services Toolkit*, is the same: to transform existing web infrastructure, namely ordinary web servers and the HTTP protocol, into a world wide networked system of modular and reusable functionality. Web Services are discussed in Sect. 2.3.

In academia, an area of research known as the *Semantic Web* (Berners-Lee, Hendler and Lassila, 2001), is exploring the possibilities and requirements of a semantically enriched web infrastructure which includes semantically tagged web pages (Payne, Singh and Sycara, 2002; Heflin and Hendler, 2001), web services (Ankolekar, Burstein, Hobbs, Lassila, Martin, McDermott, McIlraith, Narayanan, Paolucci, Payne and Sycara, 2002; Narayanan and McIlraith, 2002; Ankolekar, Burstein, Hobbs, Lassila, Martin, McIlraith, Narayanan, Paolucci, Payne, Sycara and Zeng, 2001), semantic-based matching and coupling frameworks (Frank, Szekely, Neches, Yan and Lopez, 2002; Narayanan and McIlraith, 2002; Paolucci, Kawamura, Payne and Sycara, 2002), and to some extent end user applications (Payne et al., 2002). The Semantic Web together with the Web Services architecture, form the basic infrastructure layer that will enable the service centric network. The Semantic Web is discussed in Sect. 4.4.

Finally, *Ubiquitous Computing* (Weiser, 1991; Weiser, 1994) proposes a move of the interaction model of computing from monolithic and attention demanding interfaces to distributed, decentralized, and *specialized* functions, available everywhere at any time. The Semantic Web and Ubiquitous Computing are related (Lassila, 2002) insofar as devices in the *ubicomp* world

can be regarded as services which can benefit from semantic tagging and matchmaking technology. Ubiquitous Computing is discussed in Sect. 2.1.

Very rarely, however, and to a much lesser extent, has a user focus been evident in this context. It is therefore significant that a practical service provisioning environment is provided to empower users to engage in the evolution of the service centric network; just as anyone can publish a web page, anyone should be able to publish a service. Individual Service Provisioning is a step toward such an environment.

1.4 Research Problem

The encompassing theme of this thesis is the emerging field of electronic service provisioning, and more precisely, the user's perspective of the same. The research problem is stated as follows:

> *How can we provide a technology that enables specialized and personalized electronic services to be commonly available at the right time and the right place?*

The problem is further subdivided into four complementary problems:

Interaction with services. How can we provide an environment that allows the user ubiquitous, continuous, and simple interaction with a personal set of services?

Service development. How can we enable users without programming skills to create their own services?

Service interoperation. How do we enable service interoperation between unrelated services? Partially, this entails providing a solution to the *standardization problem* (Lassila, 2002): for two unrelated services to be able to interoperate in a meaningful way, they must agree, *a priori*, on some standard for communication, interoperation, and/or semantics. Without the standard, interoperation is impossible

Service provisioning. How do we make it possible for users to provide services to one another?

1.4.1 Delimitations

Service provisioning includes areas such as delivery, infrastructure, development, and interoperability (Sun Microsystems, Inc., 2001); the user's perspective also includes service management, and interaction with services.

The areas of delivery and infrastructure include aspects such as methods for deploying services on accessible servers (i.e. application servers and web service end-points), network standards for service access (HTTP, SOAP, RMI, WAP, etc.), common formats for the packaging of the service functionality (applets, Midlets, servlets, etc.), and service environments in which services can execute on the user's computer (Java sandbox for Applets in a web browser, Midlet environment in a mobile phone, etc). The areas of delivery and infrastructure largely fall outside the scope of this thesis; most of them, however, have been touched upon throughout the present work. Nevertheless, one significant part of this thesis work falls within this category: the sView personal service environment. SView is a prototype environment allowing end users to keep, manage, and interact with their services. Part of the sView framework is also the specification of the packaging of services to make them compatible with sView.

Development of services is a broad area within service provisioning. Services can be developed much as applications are developed: using development tools and models, including programming languages, development environments, object-oriented modeling methods, to name a few. Traditionally, applications and services have been developed by a third party, usually a company or organization, and more rarely, an individual. Although the full breadth of the area of service development is outside of the scope of this thesis, one interesting aspect is covered here: the development of services by ordinary end users.

Interoperability between services is concerned with the way in which services can interoperate to perform more complex tasks. This area includes a wide range of difficult problems akin to those faced in Enterprise Application Integration (EAI) (Linthicum, 1999) and the Semantic Web (McIlraith, Son and Zeng, 2001): communication protocols, semantic agreement, data conversion, etc. The following quotation from Linthicum (1999) describes the situation in EAI:

> "The EAI architect must understand what each of these business services [that we are trying to connect] does, what the required information for each service is, and what the expected outcome is."

The present work similarly tries to achieve interoperability between services, although, in this case, it is the end user, not a professional developer, who creates the couplings.

The user perspective within electronic service provisioning is rather complex; not only does it include aspects of development, delivery, infrastructure,

and so on, it also includes traditional issues of human-computer interaction. The present work, however, is focused on the following:

- Managing and organizing services. Our approach is to provide each user with a personal service briefcase. Within the briefcase the user can view, select, insert, and remove services
- Accessing and interacting with the services. This is also done through the service briefcase. SView supports a number of means of access, for example, simple text-based access through a mobile phone interface

Next, we examine our approach to perform the research.

1.5 Approach

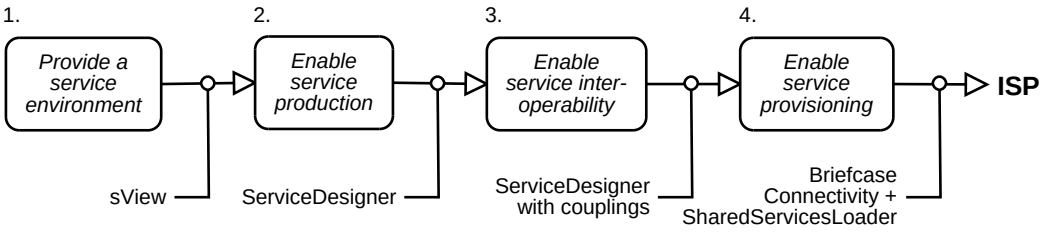

Figure 1.1: The steps to achieve Individual Service Provisioning.

To enable Individual Service Provisioning this thesis proposes the following four steps (Fig. 1.1):

1. First, provide the user with a supporting environment for interacting with and managing services. It should promote the use of services through simplicity, ubiquitous access, continuity, and user control. *SView* is designed and implemented as such an environment
2. Second, to give the user access to a significant library of services with the distinctive feature of having one or more well understood and delimited functions, Web Services[4] were chosen as the component parts; they serve as building blocks for the practical demonstration system that was designed and implemented. The access is provided by the

[4]Web Services are programmatically accessible (i.e. using computer programs) functional components that are utilized over a network. A more extensive description of Web Services follows in Sect. 2.3.

ServiceDesigner tool, which lets users simply and effectively perform the necessary preparations to create sView services from web service components

3. Third, enable service interoperability by allowing the user to combine services to create specialized and more personalized services. The resulting services should be compatible with the user's service environment. The ServiceDesigner has this capability and relieves the user from having to wait for a professional developer to create the services. ServiceDesigner, when tying together unrelated services, also tackles the standardization problem

4. Fourth, support and encourage the proliferation of great numbers of services, both specially built and aggregated (built from existing component services). This is done by sharing services between users with the *Briefcase Connectivity* system and the *SharedServicesLoader*

What distinguishes our work in this arena is the combination of a personal service environment, the possibility for users to create their own services, and the reuse of services, through the provisioning system. The reasoning follows:

The personal service environment collects regularly used services for easy and convenient access, as opposed to a scheme in which services are dynamically constructed when they are needed (see for example "The Perfect Storm" scenario in (Hendler, 2001)). This does not necessarily imply that all services must be loaded into the environment beforehand, but rather, that the services have been *created* beforehand, by another user or developer providing his or her human intelligence for the semantic understanding. This saves us from having to use complex matching and reasoning algorithms with declarative semantic descriptions (as are used with semantic web services (Paolucci et al., 2002; Lassila, 2002; Ankolekar et al., 2001)).[5] Furthermore, end users share their services with each other; thus, we enable a dynamic flow of services wherein good services propagate and bad services stagnate. Our position relative the industry and academic efforts is complementary; one part of the problem is solved by the Web Services infrastructure, another by the Semantic Web, and a third by Individual Service Provisioning.

1.6 Contributions

The main results of this work are threefold:

[5] We do, however, intend to *complement* our method with semantics (see Sect. 6.2.3).

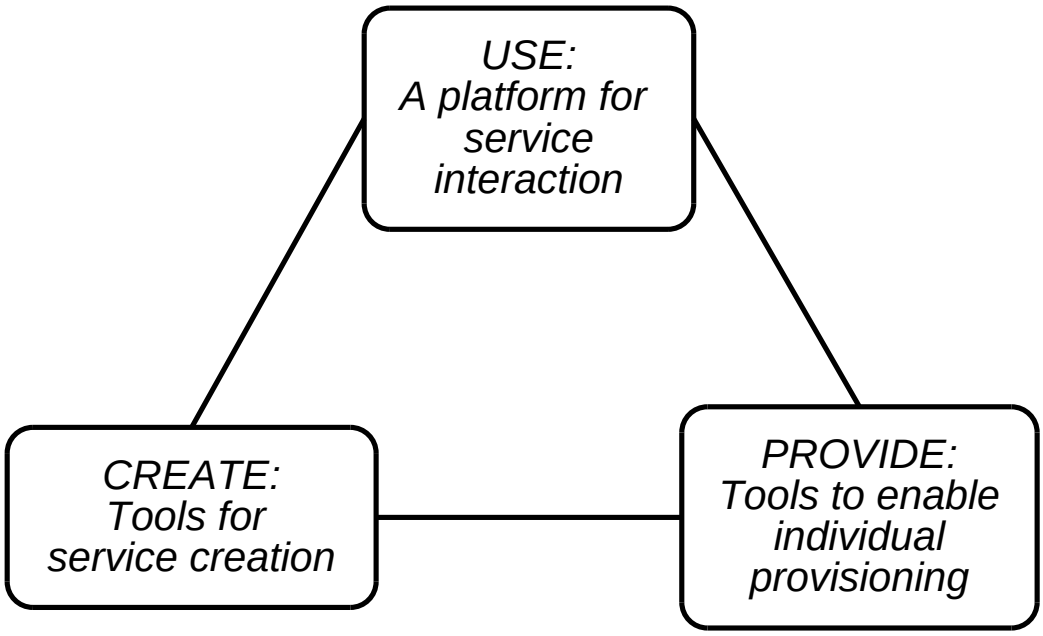

Figure 1.2: The three parts of the main contribution: *USE*, *CREATE*, and *PROVIDE*.

1. The overall design of Individual Service Provisioning
2. A set of requirements for enabling Individual Service Provisioning. These requirements are:
   - *Incentive* and *ability* to use services
   - *Incentive* and *ability* to create services
   - *Incentive* and *ability* to provide services
   - *Incentive* and *ability* to trust services
3. A set of implemented systems that in practice seeks to fill the above requirements (see Fig. 1.2) :
   - *USE*: *sView*, a personal service environment that enables a user to use services
   - *CREATE*: *ServiceDesigner*, a tool that enables users to create new services from service components
   - *PROVIDE*: *Briefcase Connectivity* and *SharedServicesLoader*, sub-systems of sView that enable users to provide services

The papers of the thesis correspond to the parts as follows: the first and second papers, Paper A and Paper B, correspond to the *USE* part; the third paper, Paper C, corresponds to the *CREATE* part; the fourth paper, Paper D, corresponds to the *PROVIDE* part; and the fifth paper, Paper E, corresponds to the whole.

The most important contribution of this work is the amalgamation of the three parts into Individual Service Provisioning. Of the total time and effort spent on this work, the greatest portion (approximately 50%) was spent on the USE part, first in the work contributing to the authors Licentiate thesis (Paper A) and then in cooperation with Markus Bylund for the sView work. Of the three parts, however, the *CREATE* and *PROVIDE* parts are most important; it is the creation and provisioning parts, in combination, that make the work unique.

This work has also contributed in a number of practical application scenarios:

***USE* and *CREATE*:** In the EU funded research project FEEL,[6] the sView system was used as a platform for designing, implementing, and testing mechanisms to manage the level of distracting interruptions, or *intrusiveness*. In various meeting scenarios sView would act as each participant's personal service briefcase, which would be brought with the user into the environment. During meetings, services in sView would be controlled by a special intrusiveness manager, called Sentinel (Espinoza and Hinz, 2003), which would negotiate between all users to achieve a consensus as to the appropriate level of allowed intrusiveness. The Sentinel and sView would also enforce this level on services

***USE*:** SicsDAIS, the precursor of sView, was used as a common user interface to multiple agents in the EU project KIMSAC (Charlton, Espinoza, Mamdani, Olsson, Pitt and Somers, 1997)

***CREATE*:** Within the Web Services community, ServiceDesigner has been recommended as a tool to quickly and simply test Web Services, by the Apache SOAP Frequently Asked Questions list (Apa, 2002)

***USE*:** In the TAP project, managed by Magnus Boman at SICS, sView is being used as a platform for conducting research regarding accessible autonomic software, i.e., the interaction of humans with autonomous software, for example in market spaces (Boman, Bylund, Espinoza, Danielson and Lybäck, 2002)

[6] http://www.dsv.su.se/feel/

***USE*:** ADAPT explores automatic adaptation in open systems, emphasizing adaptation toward user preferences and device properties (Nylander and Bylund, 2002). This work is a direct descendant of the sView work, and is headed by Markus Bylund, in the Open and Adaptive Service Infrastructures (OASIS) group at SICS.

1.7 Research Chronology

The research described in this thesis has been performed over a period of six years, from 1997 to 2002, within the Human Computer and Language Engineering Laboratory (HUMLE) at the Swedish Institute of Computer Science. It grew from the ideas of human interaction with multi agent systems (for example, Moran, Cheyer, Julia, Martin and Park, 1997) and developed into the sicsDAIS system within the KIMSAC project (Charlton, Espinoza, Mamdani, Olsson, Pitt, Somers and Wærn, 1997; Charlton, Espinoza, Mamdani, Olsson, Pitt and Somers, 1997). When KIMSAC finished in 1998, we started work on a new and improved version of sicsDAIS, which came to be called sView. This work took place within the SITI[7] funded Internet-3 project I3SVIEW.

The sView project had two main goals: to create a platform for interaction with services, and to create a platform in which to base other research. In regard to the present work, sView formed the baseline in two ways: it constituted the main contents of the *USE* area of Individual Service Provisioning and it enabled the work within the two other areas, *CREATE* and *PROVIDE*.

At this time, the focus started to shift away from the basic platform and toward the more specialized area of service provisioning. This was a natural development as the platform work had become a cooperative effort between the author and Markus Bylund, and the author wished to further evolve the original ideas of interoperating services, which had been described in the original platform work pertaining to sicsDAIS. The work on the ServiceDesigner, a part of the *CREATE* area, started early in the year 2000, and the work on Briefcase Connectivity, the main part of the *PROVIDE* area, started late in 2000. These two efforts, with sView as the base, came to be the known as the *Cooperating Services* project, a loosely tied "virtual" project within HUMLE, being influenced by topics such as *Ubiquitous Computing* (Weiser, 1993; Weiser, 1991) and the *Semantic Web* (Berners-Lee et al., 2001). From the amalgamation of this work later came a few sidetracks, into virtual location based digital notes (i.e. *GeoNotes*: Espinoza,

[7]Swedish IT Institute, http://www.siti.se

Persson, Sandin, Nyström, Cacciatore and Bylund, 2001) and context simulation (*QuakeSim*: Bylund and Espinoza, 2002), but the main thread was continuously the thesis work: *Individual Service Provisioning*.

The branching of activities was a natural one as several more participants, with differing interests, over the years were tied to the common theme surrounding sView. What started as a one-person activity to create an interaction platform for agents in KIMSAC, had at the end of the sView project grown to a group of six people working in five different projects, within the OASIS group. At the end of this period, the author was also the group leader for OASIS and project leader for the sView project and the SICS part of the FEEL project.

1.8 Overview

This thesis is built around a set of papers detailing the problems and solutions that we propose for *Individual Service Provisioning*. The first part consists of a set of chapters with the structure described below; the second part consists of the set of papers. Finally, in Appendix A, can be found a tutorial for using the *ServiceDesigner*.

Chapter 1 outlines the motivation, problem, and proposed solution of *Individual Service Provisioning*.

Chapter 2, *Background*, reports on the evolution of the World Wide Web, Web Services, and the overall increasing focus on electronic services.

Chapter 3, *Individual Service Provisioning*, describes the design and implementation of the three components of Individual Service Provisioning: the personal service platform *sView* and its services in Sect. 3.2, the service creation tool *ServiceDesigner*, in Sect. 3.3, and the service provisioning components, *Briefcase Connectivity* and *SharedServicesLoader*, in Sect. 3.4. This constitutes the main material of the work.

Chapter 4, *Summary of the Papers*, summarizes each of the papers.

Chapter 5, *Related Work*, describes other work relating to the three parts of Individual Service Provisioning.

Chapter 6, *Conclusions and Future Work*, summarizes and concludes the thesis and offers a brief look into the future of Individual Service Provisioning.

Next to last follow the five papers, and finally, the Service Designer tutorial.

Chapter 2

Background

The Personal Computer introduced a new generation of computer usage. After batch systems and mainframes, which reserved computer processing for a very limited number of sharing users, the PC represented a device that each user could put on his own desk. At first, the command line interface was the predominant interaction method but this soon changed as the *Graphical User Interface* was invented. The GUI was a revolution in early Human-Computer Interaction, but even more significant was the possibility for a single person to control his own computer.

As the PC has become standard issue in most homes, corporations, and other institutions, it has also evolved technically; in many cases, its processing capability far exceeds the processing demands of its user. Interestingly, this excessive capacity—computer processing power, disk space, and network bandwidth—has come to be harnessed collectively in experimental systems such as *Spawn* (Waldspurger, Hogg, Huberman, Kephart and Stornetta, 1992), a distributed market-based system for simultaneously utilizing idle processing power on many computers, large scale distributed systems such as SETI@home (SET, 2002), and peer-to-peer based systems such as Napster, Gnutella, and Freenet (Nap, 2000; Gnu, 2002; Fre, 2002). Notwithstanding the importance of the GUI invention, one could argue that the more important change of this generation of usage was the ratio of one user for one computer.

Today we are witnessing the beginning of the next generation of computing; computer technology, including processors, memory, interface technology, etc., has grown sufficiently advanced and inexpensive, as to warrant and make economically viable for each user to have many computing devices. Personal Digital Assistants, mobile "communicator" phones, laptops, web-pads, set-top boxes, calculators, Video Cassette Recorders, Web-TV systems, keyring encryption key generators, and other devices are becoming omnipresent.

This is the commencement of the fourth generation of computer usage, which has been called *Ubiquitous Computing* (Weiser, 1991).

2.1 Ubiquitous Computing

> "For thirty years most interface design, and most computer design, has been headed down the path of the "dramatic" machine. Its highest ideal is to make a computer so exciting, so wonderful, so interesting, that we never want to be without it. A less-traveled path I call the "invisible"; its highest ideal is to make a computer so embedded, so fitting, so natural, that we use it without even thinking about it. (I have also called this notion "Ubiquitous Computing", and have placed its origins in post-modernism.) I believe that in the next twenty years the second path will come to dominate. But this will not be easy; very little of our current systems infrastructure will survive." Mark Weiser, 1988 (Ubi, 1997).

Ubiquitous Computing is mainly characterized by two attributes: the ratio of more than one computer per user, and the disappearance of the computers. Instead of putting one computer at the center of the user's attention, many computers and computing devices are hidden in the user's environment, connected by wired and wireless networks, to become tools that can be used while focusing on the task instead of the tools.

In contrast, Virtual reality, for example, aims to recreate the physical world with the user in its epicenter; the user is an all-powerful actor with unlimited direct access to the power of the digital world. The computing environment becomes the center of attention and is very plainly distinguishable from the real world. The problem with virtual reality, according to proponents of Ubiquitous Computing, is that humans live and relate to each other in the real world; the virtual world per definition is separate to this. As another example, personal agents similarly place the computer environment in the center of the user's attention. With agents, the user transfers the tasks of performing computing operations to the agent(s); these act as servants to the user.

Weiser states: "A good tool is an invisible tool. By invisible, I mean that the tool does not intrude on your consciousness; you focus on the task, not the tool." (Weiser, 1994). We concur. In Ubiquitous Computing, the computing devices are your tools and as such, with practice and familiarity, should become part of the tasks for which they are used; everyday computing

becomes similar to reading, writing, or mowing the lawn—activities that are performed automatically, nearly unconsciously, in pursuit of enjoying a good book, expressing your feelings in a letter, or getting the grass cut.

Ubiquitous Computing involves the interconnected functioning of many distributed computing devices. In many cases, no single device will be capable of performing the necessary operations, instead the devices will connect to each other dynamically, to perform the task jointly. Some of these devices will be stationary and connected to a wired network and some will be mobile and wireless. The distribution, mobility, and number of devices, collectively places great requirements on the network infrastructure; more devices obviously increases the need for bandwidth, and the mobility of devices requires more flexible means for identifying nodes and routing data between them. The more interesting problem however, from the perspective of this thesis, are the demands caused by the interaction between the user and the computing environment.

Mark Weiser called the first efforts of research in Ubiquitous Computing for the first phase of making computing invisible. He predicted that this phase would be lengthy due to the changes that are required in the design of hardware devices and existing physical infrastructure (Weiser, 1993). Of course, infrastructure here means the network and physical device infrastructure, and as such, we are presently still involved in the first phase: the Internet is expanding to include mobile devices through wireless local area networks and cellular networks, and devices with computing ability are becoming truly ubiquitous, for example in household appliances and personal handheld computers. But what is the next phase?

If Ubiquitous Computing is to become truly ubiquitous and invisible, the devices and infrastructure must be supported by a software layer—a middleware, or *software infrastructure*. This layer has to be flexible and general enough to encompass a heterogeneous set of applications or *services*. It must allow users and services to move dynamically in and out of the computing landscape, all the while keeping track of each entity, and enabling the entities to interoperate transparently. This can be viewed as the second phase of Ubiquitous Computing (Weiser, 1993), and this is where Individual Service Provisioning is situated.

2.2 Increased Focus on Services

Computer usage is moving toward a mobile, distributed, and service centric model. The evolution of the World Wide Web indicates this, as does

the developments of wireless network infrastructure and smaller computing devices.

The Web, and the adoption of mobile telephony, has contributed to change the culture of technology (Davies and Gellersen, 2002): the Web as a global information and service carrier, and mobile telephony as an almost ubiquitous communications system. Thanks to these technologies, people are more inclined to embrace the ubiquitous computing technology and less inclined to associate it to a single computing device. For example, many users now access a single point in the digital space from any number of physical devices.

Let us consider the Web more closely. It started as a publishing system for static information, but the publication of information has evolved into the provisioning of interactive services. The Web has become a great success since it is easy to create and distribute content and it is easy to access the content. Users are now accessing their services such as banking, ticket booking, information searching, and communication over the Web. Unfortunately, the Web is not an optimal platform for services. These are some of the reasons:

1. The Web is based on a client/server model where a server supplies many clients with the same web based service. As the number of users grows, the scalability of this model becomes a problem

2. Access to services is limited to page based interfaces such as web and WAP browsers. Not only do these interfaces provide limited interaction capabilities (which are ample for publishing and accessing plain information, but insufficient for highly interactive services) but they also depend on an ever-existent network connection. If the connection breaks, a web interface is unable to provide access to the service

3. User control of web-based services is low. A service provider will many times strive to personalize a service to make it more useful or pleasing to a user. To do this, the service provider needs access to information about the user. In a web scenario, this information is transferred to the provider at which point the user's control over the information is lost. Additionally, as the number of services the user subscribes to increases, the risk of unauthorized use of the personal information also increases. Finally, as the information changes (for example if the user changes his or her address) the user must update this change with all subscribed service providers

The development of mobile computing and communication, mobile phones, PDAs, ATMs, and a wider deployment of interactive services over

broadband, wireless, Digital Audio Broadcast (DAB), cable-TV, and digital-TV networks, further reinforces the trend toward services.

Furthermore, the reasoning surrounding Ubiquitous Computing, above, suggests that the advancement will move even further; in a somewhat more distant future we move toward a computing model wherein traditional computers may disappear completely, to be replaced by intelligent artifacts powered by highly specialized computing devices.

We are already seeing signs of this development: smart rooms and houses (Kidd, Orr, Abowd, Atkeson, Essa, MacIntyre, Mynatt, Starner and Newstetter, 1999; Coen, 1999; Coen, 1998), dynamic discovery and interoperation of services (Bauer and Dengler, 1999; Rekimoto, Ullmer and Oba, 2001; Ankolekar et al., 2002; Narayanan and McIlraith, 2002; Payne et al., 2002; Paolucci et al., 2002; Lassila, 2002; McIlraith et al., 2001; Ankolekar et al., 2001; Sun Microsystems, Inc., 1999), context aware applications and services (Espinoza et al., 2001; Dey, 2000; Kohtake, Rekimoto and Anzai, 2001), and platforms for ubiquitous access to services (Davies and Gellersen, 2002; Sun Microsystems, Inc., 2001; Garlan, Siewiorek, Smailagic and Steenkiste, 2002).

The key concept in this scenario is the *electronic service*. Electronic services will be the user's points of access when interacting with the service world. Electronic services will be accessible using the new devices and they will be integrated and intertwined with the Web. The electronic services and their enabling software infrastructure are the components of the second phase of Ubiquitous Computing.

2.2.1 Agents and multi-agent systems

Another view of services builds on the concept of agents and multi-agent systems. Agents have been described as communicating software components with traits such as reactivity, autonomy, proactivity, and so on (Genesereth and Ketchpel, 1995; Nwana, 1996); multi-agent systems are collections of agents that cooperate or otherwise interact in order so solve more complex tasks. In the early stages of the present work, we used the domain of agents and multi-agent systems to focus the work on building enabling and unifying user interfaces for service components. The focus was never on the agents or multi-agent systems *per se*, but rather on the problems relating to the human user's interaction with such component based systems.

2.3 Web Services

The Web is evolving toward interactive services and as the latest step, *Web Services* enable the development of new services from smaller existing components. Web Services, as an example of a component based architecture, are an integral part of Individual Service Provisioning, and as such, deserve scrutiny.

Web services are information-providing and world-altering services, accessible as functional components exposed via standardized protocols. An integral component of a large part of industry's[1] effort to migrate toward a network and service-based model of computer usage, Web Services are also being endorsed as a complementary technology by academia (Ankolekar et al., 2002; Lassila, 2002; McIlraith et al., 2001).

Let us consider a web service scenario. We start out with a user-centered and distributed computing model, in which users will have all of their personal information stored, accessible, and protected by a set of web services. The services, which might include "myCalendar", "myWallet", "myAddress", etc., will serve as the user's main manifestation in the network. Application providers supply the user end-points: applications designed to use the services in the user's personal information store. Being web services, they are guaranteed to be accessible from any device and platform, and application developers can provide end-points of varying complexity, from simple one item displays that present a single piece of information, to complex aggregated services that tie in the user's personal web services along with other third party functional components. As a development model, web services in general move the important issues of code sharing and reuse another step forward compared to object oriented development; functionality available from third party providers via the Internet can be incorporated into applications during development.

The two most important protocols for web services are HyperText Transport Protocol (HTTP) and Simple Object Access Protocol (SOAP).[2] HTTP is the well-known protocol of the World Wide Web, which carries requests for service to and from a web browser and a web server. SOAP, a relatively new XML-based protocol currently being standardized by the World Wide Web Consortium (W3C),[3] carries structured data such as messages or remote procedure calls. The communications are coded and sent in SOAP envelopes over regular HTTP channels.

[1] For example, Ariba, IBM, and Sun Microsystems.
[2] SOAP: http://www.w3.org/TR/SOAP/.
[3] W3C: http://www.w3.org/.

Complementary to HTTP and SOAP, the Web Services Description Language (WSDL) (Christensen, Curbera, Meredith and Weerawarana, 2001), describes in a machine-readable format the capabilities of web services. WSDL, also an XML based language, describes everything that another program needs to know about a web service to be able to use it: its API, its network end-point, etc. WSDL documents are typically published via web pages along with information for the developer about the service provider, and the cost of using the service. Lately, directories and search engines for web services and WSDL documents have sprung up on the Web. However, for automatically finding and linking to appropriate web services yet another system is used: Universal Description Discovery and Integration (UDDI). UDDI[4] is a service registry architecture that enables service providers and consumers to discover each other, understand how to connect, and to share information. Thanks to the use of HTTP and existing well-known web infrastructure combined with the relatively simple SOAP and WSDL protocols, and the registry architecture of UDDI, web services are likely to make a significant impact on service development in the near future.

However, the end user should not be left out. It is important that web services are also easily and instantly accessible in the end user's regular service environment. There are two major reasons for this. The first is that end users should not have to wait for application developers to create applications that include or expose interesting web services. The second is that end users can aid the proliferation of services by creating their own new services using combinations of web services.

Thus, in respect to the present work, we first need to ensure that the personal service platform sView will allow the end user to load and visually instantiate a web service as a regular sView type service. This will give end users access to web services as soon as they are available on the Internet. Second, we need to make it possible for the end user to combine several different web services, with interconnected data flows, in the process of generating the sView service. This will empower end users to create completely new services, to their own specifications, that fulfill their spur-of-the-moment needs. These user-generated sView services can be shared with friends and published as stand-alone services along side of the developer targeted web services. This will aid the proliferation of new services. The next chapter details these steps toward Individual Service Provisioning.

[4]UDDI: http://www.uddi.org/.

Chapter 3

Individual Service Provisioning

This chapter is the main part of the thesis. Its purpose is to describe the concepts that are central to the objective of achieving Individual Service Provisioning as well as the enabling solutions. It does this by describing both the reasoning behind the system and the components of the implementation that make up the system, divided in three main parts: USE, CREATE, and PROVIDE. Since this thesis is a compendium of papers, many of the details, especially concerning the implementation, are left out of this text; a more thorough description may be found in the collection of papers.

We define Individual Service Provisioning as the creation, modification, improvement, and distribution of services by individual users for themselves or others. The purpose of Individual Service Provisioning is to create for users a beneficial electronic service environment filled with specialized and personalized services.

Services, in this context, are defined as functional software components that provide information processing or world-altering functionality.

The kinds of services we are referring to may be categorized as:

Original services. Services that the provider creates from scratch. This type of services requires significant technical skill on the part of the provider. Much of the work of creating a service is specific to the particular service and the platform on which it is created

Template services. Services made up of preconstructed service shells that providers fill with valuable content

Composite services. Services that the provider has built by combining and connecting other services to create new functionality

Combination services. Combinations of the previous

Our contribution is multi-faceted. Across all of the service types, spans the opportunity for us to provide technology to help providers with service deployment, service advertisement, and user-access to services. For original and composite services, the opportunity is to provide tools for constructing new and aggregated services and adding value to existing services or service combinations. SView provides the basic infrastructure platform for service delivery and access. ServiceDesigner provides the tools to create original and composite services. Briefcase Connectivity and SharedServicesLoader provide the basis for deployment and provisioning of services.

3.1 Requirements

To successfully implement Individual Service Provisioning, in order to achieve the goals stated above, there are a number of requirements. These are non-technical in their nature and stand as design guidelines for the technical implementation.

- It must be easy to use services. If it is not Individual Service Provisioning will fail. First, the system must facilitate service management, i.e., finding, choosing, installing, and administering services. Second, the interaction with services must be convenient; services should execute continuously so as to always be ready at hand, and they should be accessible for interaction via the user's most commonly used devices—such as a PC, mobile phone, or a web browser—and the system should always promote the most effective means of interaction. As the discussion of incentive shows below, if ease-of-use fails, it will not matter if the next two points, creation of services and provisioning of services, are acceptable or even outstanding, Individual Service Provisioning will still fail because the fundamental incentive to use the overall system will be lacking

- It must be easy to create services. By enabling end users to create services, the flora of services will increase. With a greater number of services comes greater diversity and services that can be more specialized. Let us compare this to the open source movement. Open source works thanks to the strong coupling between incentive to contribute and individual gain; users contribute to a system for what are essentially egotistical reasons (Yamauchi et al., 2000): to gain a better system for themselves. Since one person cannot build the whole, the individual's contribution is integrated with others' with the result that

all users benefit. For Individual Service Provisioning, the user that creates his own service is like an open source developer who contributes to the overall open source system. In this case, the overall system is the complete set of services to which the user contributes his service. Without support for easily creating services, the feedback loop will break: if there are no services, there is no point in using the system

- It must be easy to provide services to others and the overall set of services must be easily accessible to users. If a user creates a new service, and implies—by using it himself—that it is worthy of use, it must be shared with the rest of the user community; if one user find a service useful, it is likely to be useful to others as well. The provisioning of services should be an integral part of the whole system; if provisioning is missing, the ease-of-use and the creation of new services are to no avail

3.1.1 Incentives

Let us take a closer look at the incentives that power a system such as Individual Service Provisioning.

The first incentive is the service environment itself. If this satisfies the requirements above, it will provide the user with enough benefit to use the system—and more users implies more demand for services.

We established that personal need is a major factor in driving open source development—but is it enough? Consider a user with a need for a sub-routine that will allow him to perform some task in a particular circumstance in a certain system. To satisfy his need, the user must also be *capable* of providing the sub-routine (unless, of course, as is many times the case, someone else has already done so). Therefore, given the need and the competence to provide for it, an individual user can create his own solution. The same holds for Individual Service Provisioning: if the user requires a certain service, and the right tools are available to enable him to provide the service for himself, his problem is solved and he gets his service. This is the basic incentive to create services.

What, then, is the incentive that could drive a user to provide a service for others? There are many alternatives: the user may have to share, to some extent, in return for benefits available within a community; the user can receive credit for his work; the user can get help in difficult areas if someone else finds the service useful and is willing to contribute to it; and the user may get paid for the service. In file sharing systems such as Gnutella, Freenet, and

Direct Connect[1], the sharing of files is more or less an indirect effect of using the system, thus the incentive is related more to the user's personal gain in the system and less directly to the sharing. In open source development, in which contributors have been characterized as being "biased for action", or having a tendency to act first and then get feedback, this personal gain leads to an open culture and innovation (Yamauchi et al., 2000).

When we combine the creation and sharing, we get additional incentives besides the ones already mentioned: the enjoyment of creating services for others to use, helping friends, and gaining status.

Translating this to the world of Individual Service Provisioning, we get two more requirements:

- Each service that is created should not be final, i.e., it should be possible to augment it and improve upon it
- The creator and contributors to a service must get the proper credit for their work, i.e., it should be possible to label each service with the proper identification

Network Effects

The more contributors there are the more services there are and the greater the chance that suitable services will be available. This positive feedback loop contributes to a *network effect* (Katz and Shapiro, 1994) and the Individual Service Provisioning system is designed to leverage this effect.

The network effect implies that when a network increases in size, its value to the individual also increases. That is, as more and more resources enter the network, the more valuable the network becomes to the individual. In addition, systems consisting of relatively simple and similar components often exhibit more complex behavior because of the participation of their constituent parts. In biology, this is referred to as emergent behavior (Green, 1994). Using ServiceDesigner, Briefcase Connectivity, and SharedServicesLoader, end users can create and provision their own services and thus contribute to the network effect of service provisioning. By leveraging the existing body of services, new services can also be constructed in more and more complex combinations, resulting in an emergent behavior of sorts.

However, we must be aware of the "tragedy of the commons" (Hardin, 1968). The "commons", originally a grazing area in the middle of a village or township, is any resource shared by a group of people. The tragedy of the commons is that any shared resource will be exploited to the point of exhaustion because people will pursue self-interest in lieu of the group's interest. In

[1] http://www.neo-modus.com/

the case of peer-to-peer file sharing systems, there are some altruistic users that contribute significantly for the benefit of others, but research shows that a majority of users contribute nothing beyond the absolute minimum (Adar and Huberman, 2000). As long as users can exploit the common resources with impunity, they will continue to do so. Consequently, any resource sharing services which are built into sView must consider this possible threat and by design act accordingly. Glance and Huberman (1994) have shown that the participating group's size, degree of social integration, and the expected time interval of the period of participation determines whether individuals are inclined to *cooperate* or *defect*. If the group is large, and the social integration and the expected continuity are low, participants will defect, or refrain from contributing. And vice versa—if the group is small, the social integration higher, and the expected continuity large, participants are likely to cooperate and contribute. What, then, does this suggest for Individual Service Provisioning? Keeping the group of users small unfortunately opposes the goal of increasing the number of available services; and increasing the time interval of participation might be pointless since the participants are likely to have very little social integration. The answer could lie in the design of the system: if the cost of cooperation (i.e. contributing services or usage data) is sufficiently low, the above effects may be counteracted. Moreover, if participants are required to contribute in order to exploit the system, the effects could be further negated; this is evident in file-sharing systems where users must provide a certain amount of usable files in order to gain access to the best storage areas (e.g. Direct Connect).

Business Models

As a final note in the discussion about incentives, let us consider the monetary incentive a bit more closely. Obviously, the incentive to make money can many times be quite strong. But for this type of incentive to have any real significance, it must also be feasible to make money. This brings us to the broader question of what the appropriate business models are for a system such as Individual Service Provisioning. Until recently, only a few actors, in rather clumsy markets, have had enough power to attack the whole market space. Given the changing conditions provided by the Internet, including a greater market reach and systems such as Individual Service Provisioning, there is an opportunity, and room, for smaller actors to provide very specialized services. Since the market area will be the globe, and the actors are small and flexible, this opens up for a new breed of niched service providers. Of these, individuals are of course the extreme. At this point,

the missing piece in the puzzle is the payment method (which is outside the scope of this thesis.)

We now turn to the three parts of Individual Service Provisioning, starting with the personal service environment.

3.2 Using Services

SView (Bylund and Espinoza, 2000) is a Personal Service Environment (PSE) (Bylund, 2001; Bylund and Wærn, 2001), and the first part in the implementation of Individual Service Provisioning. It is a working prototype of a PSE system and it consists of a server that you may run on several machines, a service briefcase in which the user places his or her services, and *provider services*, which add to the capabilities of the system.

The user chooses services and places them in his briefcase. A sView service may be completely self-contained, or it may have a network connection to a back-end part on the network; this choice is left to the service provider. The fact that a service can be self-contained means that it is possible to run it locally without a network connection. A self-contained service is typically one that is relatively small and that has within itself all the functionality and data that it needs (for example a calculator). A service that uses a back-end often has a network connection to a database that is too large to move to each client (for example a search service for Encyclopedia Britannica). The briefcase is an environment in which services are constantly running and in which the user may interact with the services.

Services are downloaded as code modules and are executed in the briefcase. As the user changes his or her location (point of access), i.e., when going home from work, the briefcase is halted and moved to another server, where it is reactivated. The user may then access the services remotely using for example a mobile phone or a web browser at a public terminal. When the user gets home, the briefcase is once again moved, this time to the user's home server. The purpose of the moving of the briefcase is to keep the services close to the user; thereby the user is always able to interact with the services using the best possible mechanisms (a graphical user interface for a local service is more powerful than a web or WAP based interface to a remote service). Storing and running services in the user's briefcase has other advantages as well. The user's control over services increases because he is able to supervise the distribution of personal information to the services. Such information can be stored in a special preference service, which, as we will see below, can be utilized by services wishing to procure personal information about the user. Moreover, services are constantly active. This means

that they may perform tasks for the user even while the user is not on-line. It also means that the user experiences a continuing session. Each time the user accesses his briefcase, services are in the same state as the last time of access:[2] windows have the same sizes and locations, ongoing tasks are still active, and the preferences and settings of the user are persistent.

Services for sView are either *provider services*, which provide functions for other services, or *end user services*, which are used directly by the end user. When a provider service is loaded into a sView briefcase, it becomes available as a resource to other services. Other provider services or end user services can register to use certain such services and are notified when they become available; this implies that sView supports inter-briefcase communication.

A service interoperation system, such as this, allows a service developer to focus on core functionality and let other services provide auxiliary functionality. When developing an application or a service, much effort usually goes into creating auxiliary functionality such as GUIs, networking, notification, etc. (depending, of course, on what the core functionality is). When creating an end user service, the developer complements the core functionality with services that provide these, thereby saving time and money. SView's internal interoperation system, with provider and end user services connecting to each other, is based on these concepts.

Since all the user's services are instantiated inside the user's briefcase in sView, there is a great potential for service interoperation inherent in the platform. Thus, sView provides an environment that can greatly benefit service interoperation:

- SView can provide security mechanisms that regulate how, when, and to what extent services may interoperate
- SView can collect in one place and then visualize the service interoperation combinations, to further add to the user's control
- SView can provide the user with a rich range of user-made service interoperation combinations, namely those created by other sView users. More on this issue follows in Sect. 3.4.

To demonstrate the capabilities of sView, and to encourage users and developers of sView services, we have created a set of demonstration services. The first set are modular services that add user interface capabilities to sView:

- An HTML manager that in conjunction with a Servlet manager enables interaction with services using web-based interfaces

[2] Unless, of course, they have done some work since then.

- A Wireless Markup Language (WML) manager with the same purpose as the HTML manager but for mobile phone interfaces
- An email manager for email interaction
- A Short Message Service (SMS) manager for SMS interaction
- A graphical user interface manager for graphical user interfaces created with Java's Swing toolkit

All of these provider services may be used by end user services to provide the respective interface capabilities.

Second, we also have a set of miscellaneous services including:

- The PreferenceManager service, which handles preferences (key/value pairs) for the user and for services. Being a provider service, other services can subscribe to the PreferenceManager and look up values that they need
- The Briefcase Connectivity service, which enables sView users to connect to each other, through their services, in a JXTA compliant peer-to-peer network (Traversat, Abdelaziz, Duigou, Hugly, Pouyoul and Yeager, 2002).[3] This provider service may be used by other peer-to-peer based services such as the Sentinel service (described below)
- A Console service for administering the briefcase
- The ServiceDesigner: a service for creating new services for sView using Web Services as component parts (ServiceDesigner is described at length in Sect. 3.3
- The Sentinel service (Espinoza and Hinz, 2003). The Sentinel is a group building and intrusiveness controlling service. It automatically connects (using Briefcase Connectivity) to other users in a meeting room or in another ad-hoc meeting situation and forms a group. Within the group, the many Sentinels negotiate with one another to achieve a consensus as to what level of intrusiveness should be allowed for the present meeting. Intrusiveness in this case refers to possibly disturbing interruptions from incoming phone calls, email messages, instant messages, and so forth. Each user can specify his own preference as to the intrusiveness level in the Sentinel's user interface. When the

[3]Project JXTA: http://www.jxta.org.

negotiations are finished each Sentinel receives notification of the currently allowed intrusiveness level, and then applies this level to the other services within the user's briefcase, thus controlling their actions. Depending on the agreed upon level, a service that usually sounds a signal at an incoming message, may then only be allowed to present a visual cue

Other end user services that we have built, include an email client, an mp3 player, a calendar client, a TV-guide, etc.

The sView system is really a server architecture for executing user briefcases; it is possible to service one or several users in the same sView server. A multi-user instance of the sView server—a so called sView Enterprise Server—is used to store briefcases for users when they are off-line; it enables users to log in to briefcases remotely, using the external interaction modes provided by the interface enabling services in the briefcase. One typically runs a multi-user server in a company or institution as a service to the users. In single user mode the server is run on a user's local machine. As described above, a user's briefcase is moved between different server instances and types as needed.

In his Master's thesis entitled "Implementing Services for a PSE – an Evaluation and Performance Analysis", Boman (2000) tested the sView framework in two respects:

- Its suitability as a framework for which to create network accessible service components, as compared to traditional development of web-based services, and
- Its performance under load

The early sView version was found to be at least as effective for development as its web-based counterpart was. The briefcase server also withstood considerable load; in a series of tests, between 10 and 50 briefcases were executed in parallel in one sView instance, with only a slight performance penalty.

The sView server is publicly available in a distribution consisting of a set of Java ARchive (JAR) files, start scripts, settings files, and documentation.[4] From within sView, a user may download and use both the provider and consumer services.

[4]SView is available at http://sview.sics.se.

3.2.1 sicsDAIS: the Precursor of sView

The design and implementation of sView grew out of a set of ideas, a design, and an implementation of an earlier system: sicsDAIS (Espinoza, 1998; Espinoza, 1999).

SicsDAIS is a stand-alone Java application for simultaneous interaction with multiple services.[5] It combines the distributed interfaces called *content handlers* of networked service back-ends into one easily accessible user interface. It is a mainly graphical approach to interaction with multiple services although content handlers may employ any means and modalities for interacting with users.

The back-end services are represented in the interface by the smaller content handlers—graphically capable units of Java code that may be combined in the interface to achieve the overall presentation or interaction experience for the user (see Fig. 3.1). At any given moment in the course of interaction with services, a content handler may represent a single service, or multiple services; conversely, several content handlers may work together to represent a single service. Some content handlers may even be considered orphans, as they have no direct ties to any back-end services (similarly to self-contained services in sView). Some of these may be invisible and they act behind the scenes in sicsDAIS, performing functions such as modeling of the user or orchestration of other content handlers or services (similarly to provider services in sView.)

The sicsDAIS system provides several functions for content handlers and services:

Layout. The layout engine in sicsDAIS performs automatic layout of content handlers in the interface according to specifications from the services

Data exchange. Blackboard like data exchange is available in sicsDAIS. It includes on-change notification triggers

Event registry. Content handlers (and through these, external services) can log all events in the system (as well as react to them)

Event handling model. SicsDAIS level events are internal events caused and reacted to by content handlers and other components in sicsDAIS, effectively resulting in internal communications. Basic Java level events in content handlers can be mapped to sicsDAIS events. Thus, a third

[5] In the reference texts about sicsDAIS, the term *agent* is sometimes used as a synonym to the term *service*.

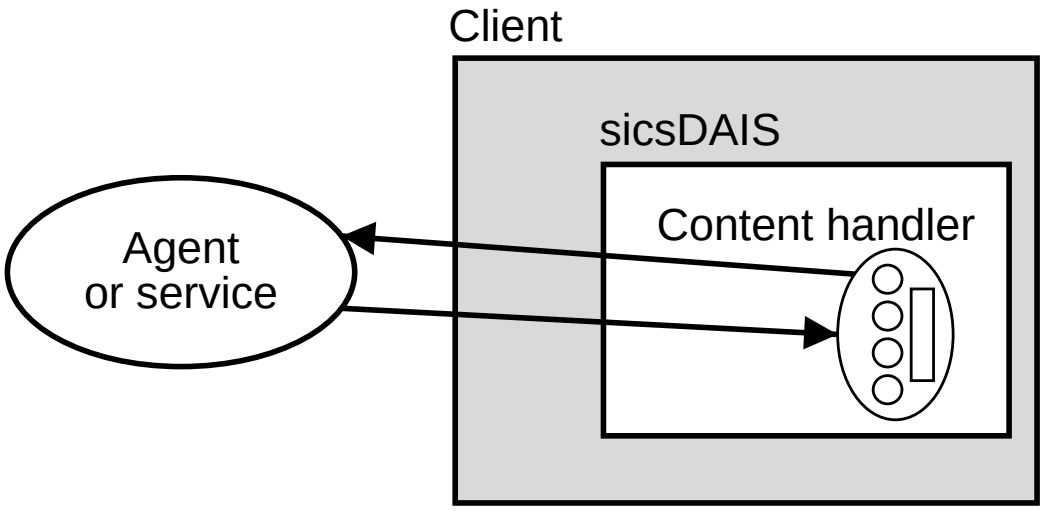

Figure 3.1: An agent communicating with a content handler in sicsDAIS, from (Espinoza, 1998).

party may use a preconstructed content handler to achieve any functionality because of an interface event in Java without modifying the Java code of the content handler; the modification is done to the script that is tied to the corresponding sicsDAIS level event. For example, when a content handler's button is pressed, it triggers an internal Java event (i.e. ButtonDown or ButtonUp). The content handler catches this event and any script associated with the event is executed. Scripts can be modified during run-time and several content handlers (and by extension—external services) can be scripted to react to such events

Exception handling. SicsDAIS provides global exception (error) handling for all content handlers. Content handlers (and services) may register to receive notification or to handle any exceptions; even exceptions of other content handlers. Thereby it is possible to create central processing of errors in sicsDAIS

Scripting language. Internal communication within sicsDAIS and between content handlers is provided using a scripting language that is interpreted at runtime. Scripts can be modified on the fly by content handlers (and thereby services)

Dynamic method invocation. (as part of the scripting language). Methods in content handlers may be called while evaluating scripts

To illustrate these features let us consider a scenario. The user has started his sicsDAIS system and wishes to interact with a service. The service connects to sicsDAIS via the network and, through a certain command, requests to have its content handler displayed in the user's interface; the user's sicsDAIS accepts and loads the content handler. The process of loading is initiated by the interpretation of a script document that specifies which content handler to load, what scripts to tie (in the event registry) to applicable events that may occur in the content handler, and basic configuration settings such as colors, dimensions, and layout, of the content handler. All of these parameters are specified in the document and the document may very well be generated by a third party, thus permitting run-time modifications to the instantiation process. While going thorough this document, the content handler is instantiated with the proper parameters, and the layout manager displays its interface. When the user interacts with the content handler, the underlying Java Graphical User Interface routines trigger events, and the associated scripts are performed. In this scenario, perhaps the user clicks a button, which results in the execution of a script that communicates back to the external service. For another instantiation of the same content handler, the button click might result in the content handler communicating with another content handler within sicsDAIS, sending it a command, in the form of yet another script, to perform dynamically.

3.2.2 The Difference Between sicsDAIS and sView

The basic concept of sicsDAIS is the same as that for sView: to bring services into a common environment in order to facilitate the user's interaction with the services. Here follows a cursory look at their differences. The main difference lies in sicsDAIS's heavier focus on presentation, i.e. the layout, display, and visual coordination of services within the environment. SicsDAIS also features a rather advanced scripting system, which allows for run-time configuration of services; sView on the other hand has extensive support for dynamic binding of services to each other, which, essentially, aims to accomplish the same thing. Lastly, in sicsDAIS, each service, through the use of its content handler(s), is responsible for interpreting its own input from the user; sView introduces the concept of *provider services* that handle different modalities and forms of interaction for all services. The inspiration for this improvement came from the Open Agent Architecture (Moran et al., 1997). Naturally, sView is a more mature and stable system, given the experience gained from the development of sicsDAIS and a more thorough system design.

3.3 Creating Services

ServiceDesigner (Espinoza and Hamfors, 2003; Hamfors, 2001) allows a user to create his own services for sView. This ability is the second component of Individual Service Provisioning.

As previously stated, the ability for users to create their own services is a very important part of Individual Service Provisioning. We believe that it is essential to have a grassroots supply of services for a service community to grow and flourish. The most obvious analogy is the World Wide Web: had the Web grown as fast and tremendously if it were not possible for individual users to provide their own web pages? Probably not. Once again, we touch upon the network effects. The Web has the appeal of being completely unrestricted and at the same time accessible with very small means; it is as easy to publish a web page, as it is to browse one. Of course, with more users, the Web attracted more attention from larger actors such as private and public institutions, and with more content being available, even more users joined.

The natural question is then: how do we make it easy to create services? Obviously, service development, just as with application development, is not something one just does haphazardly; it usually takes a great deal of skill and technical knowledge, and is therefore, traditionally, reserved for professionals. This, however, changes with the ServiceDesigner.

3.3.1 ServiceDesigner

The basic concept of ServiceDesigner is very simple. The user picks an interesting web service from the Internet, loads its Web Services Description Language (WSDL) description into the ServiceDesigner, and chooses the functional component he wishes to use (Fig. 3.2).[6]

A graphical user interface is automatically generated for the functional component, and the user, after making any desired modifications to the interface using a visual tool, then generates a sView compatible service, which is finally loaded into the user's briefcase.

The WSDL document is loaded by entering its URL into a text field in the ServiceDesigner. After loading, the available functional components are listed in the top area of the ServiceDesigner. The user chooses a functional component from this list and ServiceDesigner generates a corresponding graphical user interface.

[6]For more images, and a more thorough and hands-on description of how to use ServiceDesigner, see Appendix A: ServiceDesigner Tutorial.

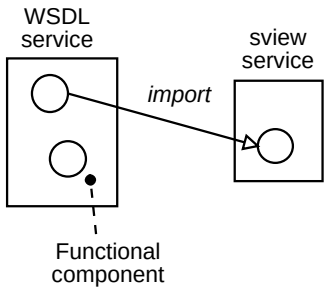

Figure 3.2: Importing a web service with ServiceDesigner.

The user can modify the interface in many ways (Fig. 3.3). For example, the user can move and resize buttons and other widgets, he can set constant values or drop down menus with variable values for text fields, and he can change the overall size of the service window. During this time, the user can also test the service by filling in values in the required text fields (which are sent as parameters to the web service), and executing the service call by pressing its execution button.

When the user is happy with the interface, he generates the service. ServiceDesigner then creates the appropriate programming code, including code to build the interface, code for making the required network connections to the web service, and code for making the service sView compatible. This code is then dynamically compiled with the result being a completely independent JAR file representing the sView service. Finally, the JAR file is loaded into sView and instantiated as a running service.

To summarize, with ServiceDesigner, an end user can easily create his own services for sView. Admittedly, the services we have considered up to this point are rather simple; they are only generated interfaces for singular web services. We now introduce interoperation between services.

3.3.2 Interoperation between Services

We define service interoperation as the process when two or more services use capabilities of all participants to jointly perform some task, calculation, or procedure. Service interoperation enables a set of services to achieve results that no single service could achieve on its own.

This particularly difficult problem merits closer inspection. There are many questions to consider: How should services publish their capabilities? Which terms should be used to describe the capabilities? How do we match the capabilities of one service to the demands of another service? Can we aid

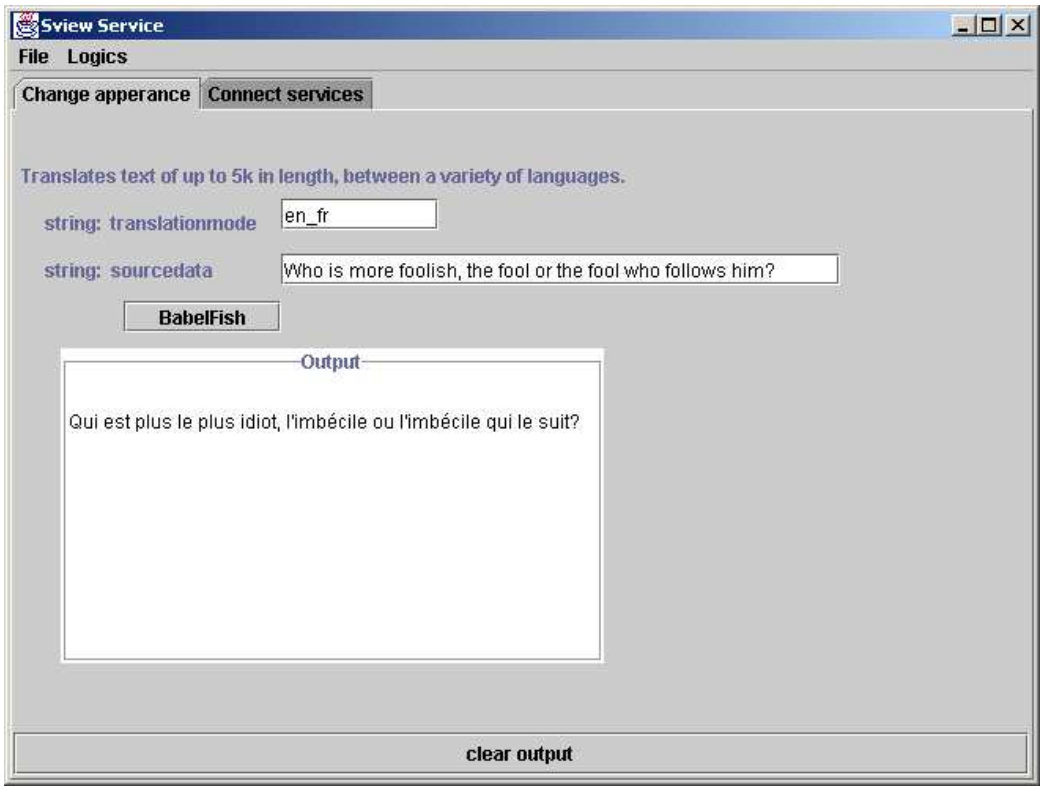

Figure 3.3: The generated GUI of a functional component from a web service.

services in transforming the output of one service to the input of another? Can the user be involved in this process? Can we benefit from other users' experiences in the process?

For a user of services, service interoperation improves the user experience and adds user control. It entails that a user can think of a new useful service and then compose it using available components; there is no need to wait for a developer to create a new service with the required properties. Moreover, a particularly good service can be used in many different combinations. For example, the same good spell checker can be incorporated in an e-mail service or a text editor and the same payment service can be used in many different payment situations. Additionally, the reused service can become even better by being used in many situations: the spell checker can learn new words (names for example) in one service combination and then reuse those words in other combinations; the payment service can withdraw the payments from the same account regardless of the interoperation situation.

Individual Service Provisioning defines three types of *service couplings*, or connections between services that allow them to interoperate: *hard couplings*, *loose couplings*, and *dynamic couplings* (Fig. 3.4). In Individual Service Provisioning, the user creates the couplings.

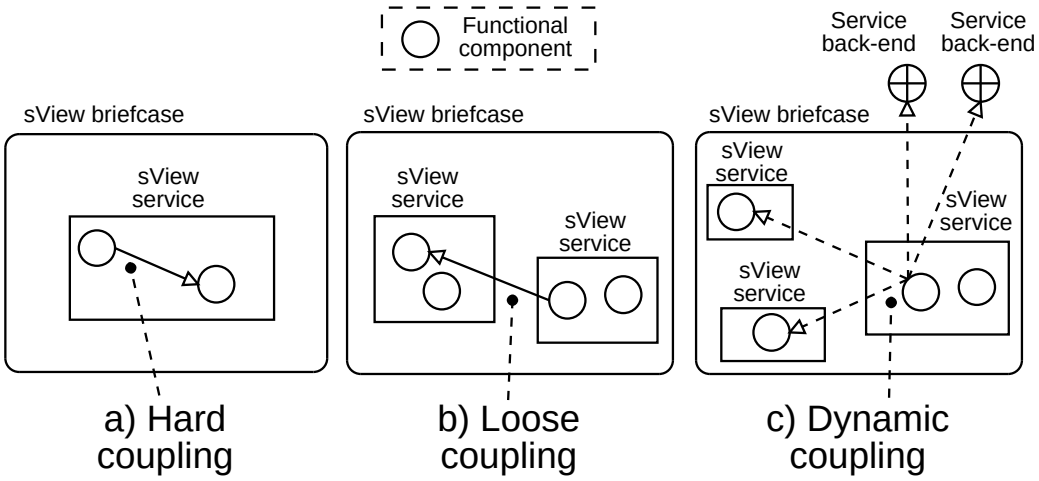

Figure 3.4: The three types of couplings.

Hard Couplings. A hard coupling (Fig. 3.4 a) connects two functional components within a service. This type of coupling is static since it is defined by the exact functional components it connects; it is analogous to traditional hard-coded procedure calls. It is used to move data from the output of one functional component to one of the inputs of another. The next section shows that ServiceDesigner is capable of tying together services using hard couplings.

Loose Couplings. A loose coupling (Fig. 3.4 b) connects two functional components *between* services in sView. The idea of a loose coupling is to move some piece of data from one service to another. Similarly to a hard coupling, this type of coupling is static, since it is the output of a specific functional component in the first service that is moved to one of the inputs of a specific functional component in the other service. This movement of data is analogous to inter-service communication. ServiceDesigner is capable of making this type of coupling between services.

Dynamic Couplings. A *dynamic coupling* can connect to either a sView service or an external service back-end, as long as they provide the same

type of functional component (Fig. 3.4 c). It connects the output of a functional component in one service to the input of *any instance* of a specific class of functional components in another service. This implies two things: first, the choice as to which actual functional component instance will be used is made at the moment of execution. Second, when the coupling is created, it is defined by the *semantics* of the specific class of functional component involved, rather than a specific instance as in the case of hard and loose couplings. This type of coupling not only requires human involvement during its creation, as the other two types of couplings, but later, when the actual execution is performed, it also requires an automatic understanding of the semantics of the coupled functional component. This implies that the functional components are described semantically and that matching can be performed between the specified class and the available instances (similarly to Paolucci et al., 2002; McDermott, Burstein and Smith, 2001). This type of coupling is more robust than the other two since it can choose from a potentially unlimited set of possible solutions; when the execution occurs, the system can try different instances until it finds one that works. Currently, ServiceDesigner is unable to create this type of coupling, but the section "Conclusions and Future Work", outlines the steps to enable this.

Let us consider an example. Assume that the user wishes to travel to London. For simplicity's sake, assume that there are only two services involved: a travel agency service and a payment service (e.g. an on-line bank). The following three scenarios illustrate the three different types of couplings.

In the first scenario, the user wants to create his own personal travel agent for buying airline tickets (he thinks it might be of use several times). He finds two web services to use: a payment service and a travel agency service. The function of the payment service is to construct electronically certified and encrypted payment parcels that can be sent to payees; the travel agency service issues tickets in return for payment. After registering a credit card and setting up an account, the payment service allows our user to issue the `pay` command with the parameter `amount`. When executed, the function returns a payment parcel, containing the specified amount of funds, which can be sent to a payee. The travel agency service has a function named `purchaseTicket`. It requires four parameters: `destination`, `date`, `time`, and `payment`. It simply transacts the purchase of the specified flight using the payment parcel as funding.[7] The user loads the description documents of the two services into the ServiceDesigner and creates a hard coupling between

[7]The travel agency service probably also has functions for searching for appropriate flights and checking for available seats but these are not immediately relevant to this discussion.

the output of the payment service and the `payment` input of the travel agency service's `purchaseTicket` function (this is described below in Sect. 3.3.3). Then he generates a sView service that automatically appears in his service briefcase. He can now enter the amount to be payed in one text field, and the other parameters in other text fields and click the "purchaseTicket" button. When the coupling is activated, it first executes the `pay` function of the payment service and then the `purchaseTicket` function of the travel agency service, with three parameters taken directly from the text fields in the user interface and the fourth parameter—the payment parcel—from the result of the `pay` function. In this scenario, the services are uniquely specified by their Universal Resource Identifiers (URIs), contained within their respective description documents. The coupling is static in the sense that it is constructed with regard to exactly these two services; it is a hard coupling because it is contained within the constructed service.

In the second scenario, the user already has a payment service running in his briefcase. It has a graphical user interface with a button for executing the `pay` function, a text field for entering an amount, and a text area, which, after execution of the `pay` function, displays the resulting payment parcel as a text string. The string can be copied and pasted into any other service that requires payment. The user now wishes to construct a personal travel agent that can receive payment from the user's existing payment service. As in the first scenario, the user finds and loads the appropriate description document for the travel agency service. He then creates a loose coupling to the payment service. This is possible since the payment service exposes the `pay` function internally within the briefcase. He generates the personal travel agent, which contains in its graphical user interface a button to perform the purchase transaction, and three text fields for entering the ticket specification. When the button is clicked the personal travel agent first executes the loose coupling to the payment service, and then, with the resulting payment parcel as the forth required parameter, executes the travel agency service.

In the third scenario, the user again has the payment service running in his briefcase. He now wishes to construct a personal travel agent, which can use *any* available travel agency service, just as long as it is semantically compatible with the required purpose. The advantage of this solution is that the personal travel agent is more likely to succeed in its effort to buy a ticket, since it can try many travel agency services until it finds one that works. The user creates a dynamic coupling between the payment service and a semantic template for a travel agency service that he finds in a travel agency *ontology*. The ontology specifies a set of standardized concepts and

terms that are commonly used among travel agencies.[8] When the personal travel agent has been created and the `purchaseTicket` function is executed, a searching and matching process is performed, which dynamically binds the personal travel agent to a compatible, and functioning, travel agency service.

We now return to the ServiceDesigner to examine its support for interoperation between services.

3.3.3 How ServiceDesigner Supports Interoperation between Services

ServiceDesigner allows completely unrelated services to understand each other—and using this understanding—to interoperate. When creating a composite service, it is the combination of the user's ability to understand the different components and their possible interrelationships, together with the ServiceDesigner's visualization ability and its connection tools, which makes this possible. Put another way, ServiceDesigner, with the help of the user, can couple unrelated software components without using any predefined specification regarding the semantics of their operations.

ServiceDesigner supports both hard and loose couplings. Support for dynamic couplings will be added in a future version.

To create hard or loose couplings in order to provide interoperation between several functional components, the user simply loads description documents for several services. As each service description is loaded, its set of functional components is displayed in the upper list of the ServiceDesigner (Fig. 3.5).

Each service that is to be involved in a loose coupling must expose its capabilities within sView. This is done using a WSDL description that is registered with a service called the *SoapPublisher*. If a service has been generated with the ServiceDesigner it has this capability built in. When the user selects which functional components to use, he can also browse and choose from the services within sView.

The user then picks the functional components he requires from this list and they are added to the area below. In the next step, ServiceDesigner generates the graphical user interface. The layout will be quite similar to that in Fig. 3.3 which contains the layout of one functional component; the difference being that there are now several sets of buttons and text fields, one for each functional component.

[8] "An ontology is a formal, explicit specification of a shared conceptualization." (Gruber, 1993).

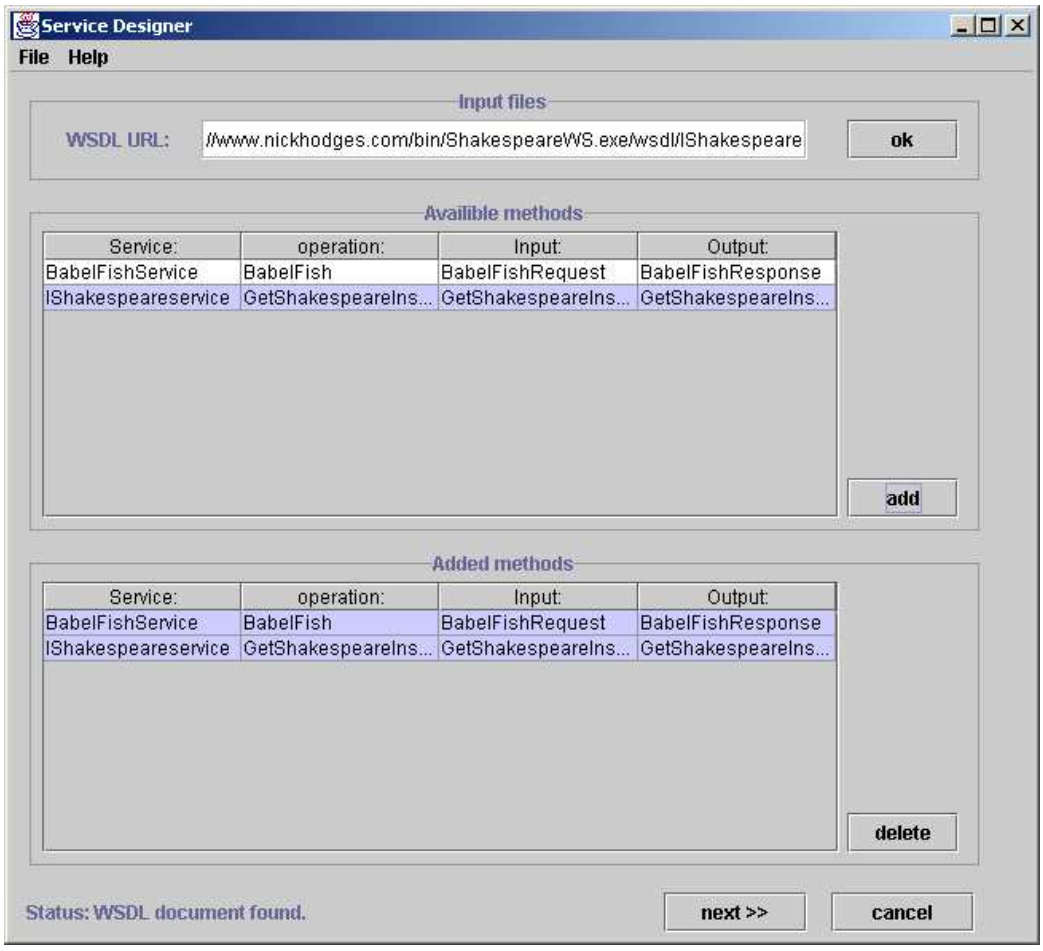

Figure 3.5: Selecting functional components from several services.

At this point, the user could generate a sView service with this layout. The service would contain several buttons, one for each functional component, and might be useful for gathering several related functions in one interface. There would be no interoperation between the functional components however; to enable interoperation the functional components must be coupled.

Couplings between functional components are performed in a second pane of the ServiceDesigner, shown in Fig. 3.6. To create a coupling between two functional components A and B, representing the flow of data between the output of A and one of the inputs, (B^1, B^2, ... B^n) of B, the user clicks and drags the mouse pointer from A to B. As B may have several input fields, however, ServiceDesigner prompts the user, by displaying a choice

dialog, to choose one from the set. The dialog lists the possible choices with a short descriptive label for each. The labels are taken from the WSDL description of the functional component. The user then selects one parameter that makes sense and the connection is complete. ServiceDesigner represents the connection with a graphical arrow.

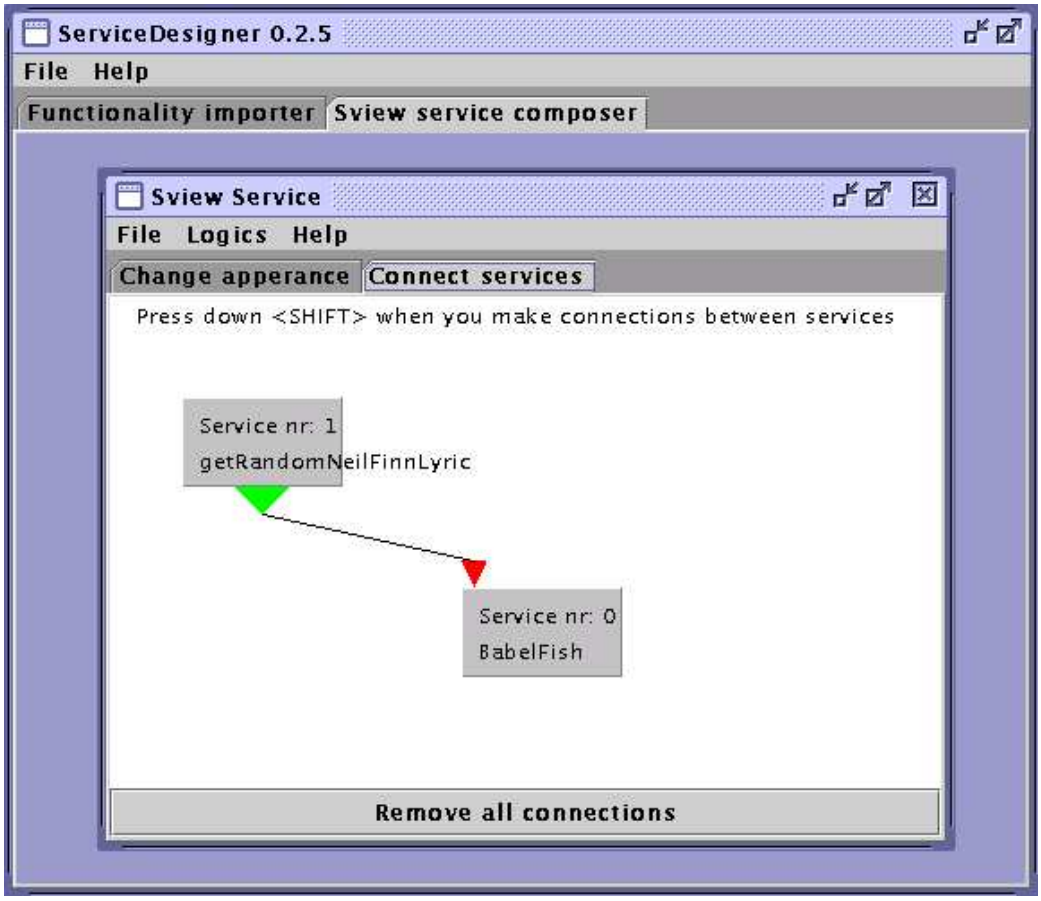

Figure 3.6: Connecting two different functional components.

When a connection is made it affects the graphical components (that were created to interact with the service) accordingly: If A is connected to the parameter B^n of B, the graphical component that represents B^n will disappear because it becomes superfluous (the user does not have to write in the value; it is taken from the output of A). Moreover, when A is connected to B, the button that invokes A will disappear. This happens because the execution of B will indirectly invoke A.

We also need some logic to handle situations where connections are not allowed. This logic should function as follows: If there is a connection be-

tween A and B, i.e. the A output is connected to a parameter of B, the user should not be able to make a connection the other way around i.e. from B to A (direct feedback), because this would lead to an infinite loop; if A is connected to B and B is connected to C, neither C nor B can be connected to A (indirect feedback in the C case and direct feedback in the B case). This would lead to an infinite loop; and finally, only output from one functional component can be connected to a parameter, when a connection is made the parameter is considered occupied.

It is the logic capability of the user that dictates if the connection will work or not—the system will not stop a user from making illogical connections. It is, however, a simple matter to test connections to make sure they work properly, and consequently, in particularly difficult cases, trial and error can always be used.

ServiceDesigner has been tested with real users in a laboratory environment (Espinoza, 2002). Two groups of users, a test group and a control group, were given three tasks involving the creation of three different combined services. Users constructed each service with two to four different functional components, between which they created hard couplings. Prior to testing, both groups were given instructions as to the functioning of ServiceDesigner and the test group was also given a description of a metaphor designed to invoke a certain *mental model*[9] appropriate for using the ServiceDesigner. The tests found that users were able to perform the three tasks and that the test group, using the metaphor, performed better than the control group. The implication of this is that for a novel and uncommon system such as ServiceDesigner, a metaphor helps users.

By enabling users to produce sView compatible services with ServiceDesigner, we increase end users' freedom of combining, individualizing, and personalizing service based functionality. More services imply more benefit to users and with the positive feedback of a growing service base, we may well see end users becoming individual service providers.

To conclude the discussion about creating services, let us reappraise the questions stated earlier:

- How should services publish their capabilities and which terms should be used to describe them? Moreover, how do we match the capabilities of one service to the demands of another service? Individual Service Provisioning depends on WSDL descriptions to publish and understand services' capabilities. The semantics of a functional component can only

[9]Mental Models have been defined as "mechanisms whereby humans are able to generate descriptions of system purpose and form, explanations of system functioning and observed system states, and predictions of future states" (Rouse and Morris, 1986).

be understood by the human operator who constructs the composite service. This means that any descriptive terms can be used as long as humans understand them. The section "Related Work" takes a closer look at how this can be done automatically

- Can we aid the system in transforming the output of one service to the input of another and can the user be involved in this process? This is exactly what is done in the ServiceDesigner; a user creates the couplings between the services, and if the output of one service does not fit with the input of another, the user can add a third—a translator service—in between

- Can we benefit from other users' experiences in the process? When a user has created a composite service, using other services as components, ServiceDesigner generates and saves it as a new service. First, this service can be used as a functional component in yet another service combination, as it exposes a new WSDL-based interface that reflects the newly created functionality. Second, this service can be provided to other users thus sparing them the effort to create the service themselves. This is the topic of the next section.

3.4 Providing Services

The ability to provide services for other users is the third component of Individual Service Provisioning.

SView enables provisioning in two ways: by exposing service capabilities as Web Services, and by providing created services in a peer-to-peer based community of all sView users.

ServiceDesigner has a built-in feature that directly enables a user to become an individual service provider; it gives each service that it generates its own WSDL described interface. After the service has been generated and it is loaded into sView, it can expose a new WSDL interface that corresponds to whatever parameters and functions the newly created service exhibits. This feature enables the sView briefcase to act as a web service end-point exactly in the same manner as other Web Service engines. This feature is also used for making loose couplings (recall Sect. 3.3.3).

3.4.1 Briefcase Connectivity

The provisioning of actual generated services builds on the provider service *Briefcase Connectivity* (Espinoza and Hinz, 2003; Hinz, 2002). Briefcase

Connectivity is a flexible and effective sView service that brings peer-to-peer technology to the sView platform. It is based on Project JXTA technology, which provides low-level discovery and networking functions.

Offering a generic messaging protocol, a peer discovery mechanism, and security features, Briefcase Connectivity allows sView service providers to concentrate on the core functionality while designing and implementing new peer-to-peer services; Briefcase Connectivity provides the peer-to-peer mechanism, which can be incorporated into a sView service using the provider/end user communication system described in Sect. 3.2. To use Briefcase Connectivity, a service in sView layers its own service specific protocol onto Briefcase Connectivity's generic protocol. Then it uses the supplied message sending and receiving functions of Briefcase Connectivity to pass messages to and from other users running the same service.

Briefcase Connectivity is currently being used in a number of different sView services:

- *SViewCommunicator* is an instant messaging system similar to ICQ. A user who runs this service can see a list containing the names of all other on-line users. With this service, users can send messages and files to one another
- *SViewHelp* is a help service specifically for sView. In this service, users can enter questions that automatically appear in other users' SViewHelp services. Any on-line user can comment on or answer questions and all comments and answers are collected together with the question
- *Sentinel* (described in Sect. 3.2)
- *SharedServicesLoader* is the peer-to-peer based service provisioning service, described below

Short Background of Peer-to-Peer

In recent years, the success of large-scale peer-to-peer file sharing applications has compelled us to rethink how we design Internet applications. Peer-to-peer applications enable users and their devices to interact with each other directly, eliminating the need for web servers and other forms of centralized arbitration. One of the more interesting aspects of today's peer-to-peer systems (e.g. Gnutella,[10] Freenet,[11] and OceanStore[12]) is the fact that they

[10] http://gnutella.wego.com
[11] http://freenet.sourceforge.net
[12] http://oceanstore.cs.berkeley.edu/

take advantage of the computing resources available at the edge of the Internet. Users confined to the Internet's edge are typically passive consumers of information and services despite the fact that their machines are grossly overpowered for the task of running simple client applications like the web browser. Peer-to-peer empowers these edge users, providing them with an intuitive way to act as both consumers and producers of information and function.

3.4.2 SharedServicesLoader

SharedServicesLoader performs the provisioning of services in sView. When the user creates a service using ServiceDesigner, its generated JAR file is stored, locally in a special directory, on the user's computer. The SharedServicesLoader looks into this directory and creates a list of all the services contained therein. When the user is on-line, SharedServicesLoader makes this list available over the peer-to-peer network (using Briefcase Connectivity) to other users of the service. In its display, SharedServicesLoader lists all other services, available from other users to whom it is connected over the network. The user can pick services from this list, and by doing so, download, and execute the services, locally, in his own briefcase. In the list, each service is described by some keywords, the author's name, etc.—items which were entered when the ServiceDesigner generated the service.

By using SharedServicesLoader, users can access services created by the whole sView community. In the current implementation, however, a user has no way of knowing either the quality, reliability, or security of a service beforehand; the only way to gain any of this knowledge is by actually downloading and trying a service. Obviously, this space affords many opportunities for improvement, and some of these are outlined in the future work section.

Chapter 4

Related Work

The ultimate goal of Individual Service Provisioning is to supply users with more specialized electronic services. We argue that this can be accomplished by supplying users with three basic components that form a hotbed for growth: an environment for interacting with services, tools for users to create their own services, and a community in which services can be provisioned.

To our knowledge, there is no other single effort that takes such a holistic approach to achieve this goal; the closest relative is the Semantic Web, which we examine last. There are, however, alternative attempts to solve the parts of the problem.

4.1 The personal service environment

The most obvious example of a related platform for service interaction is the World Wide Web.

4.1.1 The World Wide Web

In many ways, the Web is similar to sView: its openness allows anyone to build tools for access and distribution (browsers, plug-ins, web servers, etc.) and anyone can provide and consume its resources. SView is open to many types of services—and Application Programming Interfaces are freely available. The Web allows users to access content remotely. The same goes for sView. In addition to local access, remote access is possible using the Web, a mobile phone, or SMS. A major strength of sView, however, is its support for off-line activities; since services can run locally, a user is less dependent on network connectivity. Of course, the complete opposite is true for the Web: if the connection to the Internet fails, one is left stranded.

The Web is based on standards such as HTTP, HTML, and XML. SView is built with Java, which although not standardized (yet), is commonly available and close to being a de facto standard. SView also builds on HTTP, WML, and SMS, for remote access, and uses SOAP and WSDL for component service access. Overall, sView provides a super-set of functionality vis-à-vis the Web.

4.1.2 Other Related Systems

A number of other systems take steps toward gathering services in one location.

AppliGo. AppliGo, the client side of Appear Networks's Service Provisioning Server,[1] allows users to download and install location dependent services on a handheld computer. As a user enters an 802.11 wireless network that has been configured to provide services using a provisioning server, the available services' icon representations appear on the device screen. The user clicks an icon to download, install, and execute a service, and services are available for as long as the user stays within the coverage area of the pertinent wireless network. Although services do gather in one place (the user's device), there is no containing environment, such as sView, to provide common functions to the services; therefore, it is not possible for services to communicate, share resources, or interoperate in this system. SView handles loading of services by way of *loader services* within a user's briefcase, and it is quite straightforward to create a loader for location-based services. Regarding the abilities afforded by an enclosing environment, sView facilitates all of the above.

WebStart. Web Start,[2] from Sun Microsystems, allows a user to download and install Applets (small downloadable services distributed over the Web) and save them on a local machine. This greatly increases the utility of Applets, as users can download an Applet and then forget about the site from which it came. This, however, is the only real benefit of Web Start, as it in principle has the same purpose as AppliGo: to enable convenient gathering of services.

Java Services Vending Machine. A similar approach, although relating more to mobile service distribution, is described as the Java Services

[1] www.appearnetworks.se

[2] http://java.sun.com/products/javawebstart/

Vending Machine (Sun Microsystems, Inc., 2001). It promotes a model of distribution of services to mobile devices, which is very similar to both sView and AppliGo. It mainly describes the processes of finding, choosing, downloading, installing, and paying for services. Many concepts relate to sView: for instance, the delivery model enables a service to be constructed from a client-resident and a server-resident part; in sView services can also be partitioned in client and server parts, between which any sort of communication may occur. As opposed to sView, this model does not describe the final step in the chain of processes leading from service production to service consumption, namely the end user service environment. Granted, the service environment is specified as being a Java virtual machine, and the process of getting the service component into and running in the environment is specified, but the approach suffers from the same deficit as AppliGo in that the environment does not support any sort of interchange between services.

The Open Agent Architecture. The Open Agent Architecture (Cohen, Cheyer, Wang and Baeg, 1994), is a framework for integrating a community of agents in a distributed environment. The interesting similarity lies in the OAA's effort to tie together many agents in one point of interaction toward the user. Agents communicate using a declarative Interagent Communication Language, via a facilitator, a central component that arbitrates between agents. The user communicates with the system using a graphical user interface that accepts input in a number of modalities and forwards requests for translation and service to the facilitator. The facilitator analyzes requests, distributes partial tasks to registered agents, collects the results, and, finally, constructs the result for the user. Similarly to sView, the handling of specific interaction modes is off-loaded to the corresponding agents (speech input is processed by a speech recognition agent, for example); sView has its provider services that handle, for example, Web and mobile phone access, centrally, for all other services. SicsDAIS, as we mentioned in Sect. 3.2.1, lacks this capability.

The Teleport system. The Teleport system (Bennett, Richardson and Harter, 1994) enables users to bring X-based sessions from terminal to terminal in one continuous session.[3] The idea is to be able to start work in one location, say the office, go to another location, say the

[3] X (Scheifler and Gettys, 1992) is a windowing system based on servers and clients. The server runs on the user's terminal, rendering the interfaces of the clients, or applications, running on a host computer (possibly one and the same).

home, and continue where work left off. It is based on a specially constructed X server which acts as a proxy between the user's clients (applications) and the currently used X server. With this system, a user can reach his applications from any X enabled terminal. Of course, X is inherently based on the concept of a thin tier, which provides the interface rendering, and a centrally located computer, on which the actual applications run; this implies that one needs to have a network connection between the two. This model of use is possible with sView as well, however, in addition to this sView allows users to bring their services along, in a laptop, a hand held computer, or by downloading them to any Java enabled stationary computer. With sView, services run locally, as close to the user as possible, to permit a powerful mode of interaction as well as off-line use.

Next, we discuss work that relates to creating new and combined services.

4.2 Creating and Connecting Services

The ability of ServiceDesigner to automatically generate user interfaces for web services is not completely original; it is in some cases available in Integrated Development Environments, for example CapeStudio, by CapeClear.[4] The combination of ServiceDesigner and sView, however, does sport a novel benefit, namely that of creating one's own services for instant use in one's own service environment. Nevertheless, ServiceDesigner's ability to connect services is more interesting to relate to other work.

First let us recapitulate its core function. With the help of a human user, ServiceDesigner lets completely unrelated services understand one another, and using this understanding, interoperate; ServiceDesigner makes it possible to couple unrelated software components without using any predefined specification regarding the semantics of their operations.

There are other approaches to this problem that consider, and depend on such semantics; we describe these below in the section about the Semantic Web. Here we consider systems that attempt similar feats without using semantics.

4.2.1 Jini

Sun Microsystems's Jini architecture[5] allows services to dynamically discover, connect, and interoperate with one another. First, services register with a

[4] http://www.capeclear.com/

[5] http://wwws.sun.com/software/jini/

lookup registry, which they find by a method of discovery in the network; for each service, its name, capability (in the form of a Java interface), and optionally, a set of arbitrary attributes, is saved in the registry. These services act as resources in the network. Then, when another service needs to use a resource, it queries the registry with a template describing its need, and if a service that matches the query is found, it connects directly to the service and performs the transaction (to consume the resource). In this context, the discovery mechanism is interesting but irrelevant; what is relevant is the process of matching one service's need to another's advertised capability. In Jini, there are two ways to do this, both of which are automatic: either by using the attributes, or by using the Java interface. The querying service can assemble a template attribute object that contains a set of attributes that have to exist. This template is matched against the sets of attributes advertised in the registry. If a match is sufficiently close, a positive result is returned. For the interface method, the querying service simply supplies the name of a Java interface in its query and if that exact interface is registered in the lookup service, its corresponding service identifier is returned. Regardless of which method is used, to connect and make use of the found service, the querying service and the found service must agree on the interface, and this agreement must have been made in advance. Compared to the method used in ServiceDesigner, whereby services can be coupled at run-time, since the user is assisting in the process, the Jini method can be said to be suffering from the *standardization problem* (see Sect. 1.4).

4.2.2 Other Related Systems

Other systems also suffer from the standardization problem, albeit in other contexts:

- The Context Toolkit (Dey, 2000), within the area of Ubiquitous Computing, creates software wrappers around sensors, and allows a developer to aggregate these to provide higher-level context information for applications and services. All involved sensors, wrappers, and applications must agree, a priori, on the language and the semantics involved in their interoperation

- InfoPoint (Kohtake et al., 2001) is a system that uses the metaphor of drag-and-drop between real-world devices. The devices are connected over a network and data is transferred between them as the user points and clicks the InfoPoint device. For the system to work, all device capabilities need to be known beforehand

- DataTiles (Rekimoto et al., 2001) is a system of transparent plastic tiles that are placed on a reactive display device. When a tile is placed on the display, the area directly under the tile lights up with a graphical user interface of a service corresponding to the function assigned to the tile. Several tiles can be placed adjacently, and by dragging a pointing device between them, they can be made to interoperate (as a composite service). For example, if a video player tile is placed next to a tile representing a video projector, the image being played is projected on a wall instead of being displayed in the player tile. Again, all tiles need pre-configured knowledge of each other's capabilities, in this case in the form of conjoint Java interfaces

The semantic web technologies, as we will see below, attempt to circumvent the problem, and so does the ServiceDesigner.

4.2.3 InfoBeans

A system that is conceptually similar to ServiceDesigner in combination with sView is InfoBeans (Bauer and Dengler, 1999). It lets end users configure their own individualized information services from different web sites.

An InfoBean is a container that holds a small part of an existing web page. The user selects which part of the page the InfoBean should hold. The system then trains itself with the help of the user to understand how to parse out the right information even though the page has changed i.e. to learn the structure of the web page and therefore be able to understand how to handle the category of web pages that it represents.

By collecting several infoBeans in an infoBox (a DHTML[6] based web page), the information from several web pages can be gathered. Every infoBean has input and output channels, which makes it possible to connect infoBeans to transfer data from one infoBean to another.

The InfoBeans system is similar to ServiceDesigner with sView in that both systems use independent services that can be combined in one common graphical user interface. Nevertheless, a few things differ. The most fundamental difference is the choice of functionality. InfoBeans uses heuristic methods in order to find the wanted content in the web page; ServiceDesigner uses strictly defined and well-described web-based functionality,[7] which leads to a more reliable system. Second, we do not have a "server-side" that needs to process heuristic methods. This can lead to scalability problems if you cannot distribute the processes on the "client-side." Third, we dynamically

[6]Dynamic HTML.

[7]The format, or syntax, but not necessarily the semantics, are well-defined.

build a graphical user interface to the parameters of the service—the InfoBeans system renders HTML in small windows, which leads to a system that bears resemblance to many small and simultaneously open browser windows. We have total control over the actual components and can therefore change and arrange them freely.

Another difference is that the InfoBeans system lacks an overlying framework like sView. When using sView we can have other services, not just simple information services, open at the same time.

4.2.4 Enterprise Application Integration

To conclude this passage, we will briefly mention Enterprise Application Integration. EAI is described as the activity of integrating business processes within and across enterprises. In the literature (e.g. Hellman and Hellman, 2002; Linthicum, 1999), it is made blatantly clear that also EAI struggles with the standardization problem. In the context of integration between companies, however, it is unlikely that a solution such as ServiceDesigner (in its present form) would be applicable; the requirements of rock-solid performance and stability are just too great. For less mission critical applications, however, such as intranet-based information tools, it could very well be suitable.

4.3 Providing Services

Providing services with *Individual Service Provisioning* is like publishing web pages: you design a page (or design a service), publish it to a web server (or release the service in the sView community), and then view the page using a web browser (use the service in sView). The actual provisioning part of Individual Service Provisioning is very similar to what has come to be known as peer-to-peer file sharing.

4.3.1 Peer-to-Peer Systems

Peer-to-peer systems can be partitioned into three groups: centrally coordinated, hierarchical, and decentralized systems. Centrally coordinated systems, like Napster (Nap, 2000), and instant messaging applications such as ICQ (ICQ, 2002), have a central point of coordination. It relieves the distributed nodes from some of the work, for example of indexing files, and acts as a rendezvous point to connect the network. Hierarchical systems (for example Domain Name Service (DNS)) federate these responsibilities,

and nodes form groups around the coordinators in the tree. Communication between nodes around a leaf coordinator go through this coordinator, and communication between leaves go through higher-level coordinators. Decentralized systems have no coordinator and nodes form groups by means of discovery. These systems are the most interesting thanks to their scalability, robustness, and self-sustaining properties. Some examples of decentralized systems are Gnutella (Gnu, 2002), Freenet (Fre, 2002), OceanStore (Bindel, Chen, Eaton, Geels, Gummadi, Rhea, Weatherspoon, Weimer, Wells, Zhao and Kubiatowicz, 1999), and JXTA (Traversat et al., 2002).

SharedServicesLoader, the provisioning service in Individual Service Provisioning, uses Briefcase Connectivity (which is based on JXTA), as its message passing mechanism, and is thus decentralized. It works much the same as Gnutella: users' services are listed within, and when another user performs a search for a service (or rather, activates the loader, which is equivalent to searching for everything), all connected nodes respond with their lists. The lists are integrated and the available services are presented to the user. In a future version, the SharedServicesLoader will also provide more fine-grained search capabilities.

4.4 The Semantic Web

The Web of today is designed for human usage; the markup language HTML is mainly a markup for content and presentation for humans. This is fine for displaying information for humans to read but it is difficult for programs and machines to interpret. The usual remedy to this problem is to use contemporary search engines that index web pages according to their contents and allow users and programs to search for relevant material using key words. The meaning, or *semantics*, of the contents, however, is only understood by the human interpreters. There are also web pages that present functions in addition to information; among these are shopping sites, sites for database retrieval, etc. Performing the functions of such web-based services involves interaction between the underlying logics of the site and the user. For a program, the lack of semantics of these sites presents a major problem. As a makeshift solution, it is possible to create special wrappers that present programmable Application Programming Interfaces (APIs) to a site. These wrappers manually extract and insert the appropriate parameters into the site when called by an external program;[8] if the site changes its layout or design, however, the wrapper most likely has to be reconfigured by hand.

[8]This is sometimes referred to as "screen scraping."

The Semantic Web, as proposed by Berners-Lee et al. (2001), aims to evolve the World Wide Web to include semantics. With a semantic web, programs will be able to understand the purpose, contents, and interactions of a web page. The Semantic Web will include ontologies, i.e., repositories of concepts and relations pertaining to the contents and function of the sites (Gruber, 1993). Using ontologies and the Semantic Web markup languages (for example DAML and DAML-S (Ankolekar et al., 2002; Hendler and McGuinness, 2000; Ankolekar et al., 2001)) sites can define their contents and function in agreed upon terms that establish their relationships; domain independent ontologies for defining terms such as "service" can be sub-classed by domain dependent ontologies for specialized services such as "BuyBookService" and "AmazonBuyBookService."

Using the Semantic Web, developers can create programs that can do more advanced processing of the available information as well as automatically invoke the services that the sites contain. Advanced agent-based processing of Semantic Web enabled sites will include automatic discovery, execution, composition, and interoperation of services (McIlraith et al., 2001; Ankolekar et al., 2002). This area of Semantic Web research is largely related to the present work. To illustrate the similarities and differences we consider the paper "Semantic Web Services" (McIlraith et al., 2001), wherein the authors state:

> "We argue that many of the activities users might wish to perform on the Semantic Web, within the context of their workplace or home, can be viewed as customizations of reusable, high-level generic procedures. Our vision is to construct such reusable, high-level *generic procedures* and to represent them as distinguished services in DAML using a subset of the markup designed for complex services. We also hope to archive them in sharable generic procedures ontologies so that multiple users can access them."

Let us compare the Semantic Web with Individual Service Provisioning. To start with, the Semantic Web covers a greater area of research. It considers the coding languages, the reasoning engines, the matching algorithms, the ontologies, etc., needed to enable semantic processing of web-based information and services. The most relevant parts of the Semantic Web research, as it pertains to Individual Service Provisioning, are those concerning semantic coding of services and the tools to match, combine, and reason about services.

This is how our work is similar and different from that of the Semantic Web:

The vision. The overall purpose of the Semantic Web service research is to enable users to get access to personalized and specialized services, i.e., the right service at the right time; in this respect, our work has the same ultimate goal. The Semantic Web research is, however, more focused on the lower level "plumbing" that is needed. We are more concerned with the situation of the end user, an area that is not, as of yet, considered at any great depth by the Semantic Web research. Similarly to McIlraith et al. (2001), the services we aim for are mostly very simple in terms of interactivity while they at the same time are able to provide a great deal of benefit to their user.

Creating services. Instead of providing developers with the tools to create services from scratch we propose to empower users to assemble the required parts into working services using the ServiceDesigner.

Service composition. Our concept of service composition entails connecting services together to create interconnected data flows; the output of one service becomes the input of another. The resulting service corresponds to the *generic procedure* and is made possible by the user involvement. McIlraith et al. (2001) compose a service from more basic pieces, not necessarily *connecting* them but rather *collecting* them with the preconstructed generic procedure as the template. Narayanan and McIlraith (2002) describe automatic service composition, as well as simulation and testing of such composites. To a certain extent, an individual user is capable of testing his own services, and testing becomes a statistical measure when one can track the usage of services. If a service is available in the community and it gets a great deal of usage, this can be taken as an indication of its correctness.

The semantics. To enable services to be understood by the deductive machinery, McIlraith et al. propose to mark up services with declarative data, meta-data, properties, capabilities, interfaces for execution, prerequisites, and consequences of their use; this enables automatic web service composition. Our approach is more simplistic in terms of the semantic markup but conversely relies more heavily on the abilities of the human service creator in understanding the component services; this enables individual human service provisioning.

Providing services. We also enable users to provide services through the Briefcase Connectivity peer-to-peer system; these services, however, are not only created by professional service providers but also by users, for users.

Our approach results in very specialized services, adapted to individual demands, and available to many users through the provisioning system. The advantage of this scheme is that we do not have to rely on service developers to create the services that users demand. Even with automatic composition of services, which according to Lassila (2002) (in contrast to our opinion), will make user involvement in the process unnecessary, our solution should still be viable. Lassila does make a valid point (again in our opinion) about semantic services in a "volatile" environment:

> "not only can any device or service fail or be removed from the environment at any time, but new ones can be added to it: opportunistic exploitation of services might thus be beneficial"

With ServiceDesigner, the couplings made between the constituent components of a service are "hard," i.e., if a component fails or is removed from the network, the encompassing service will also fail. This is, however, a conscious design decision to promote simplicity in ServiceDesigner. In a future version, we may add the possibility of specifying replacement services for cases when the ordinary services fail.

A second disadvantage of our system is that we do not make use of semantics to discover services. This prevents us from doing automatic matching between the needs of a user creating a service, i.e. which building blocks to use, and the available services in the network (e.g. Paolucci et al., 2002). We instead rely on the user to find the appropriate services using other means (such as simple key-word based search, and meta-level services such as top-lists, review services, or recommender systems). Our system would no doubt benefit from the addition of semantic discovery and matching capabilities as we outline in Sect. 6 (Conclusions and Future Work), as well as facilities for service execution monitoring, specification of pre-conditions and post-conditions for use, and quality guarantees, as pointed out in (Ankolekar et al., 2001).

Finally, with sView, we also provide to users an encompassing environment in which to gather and interact with services, which in turn leads to more efficient and coherent utilization of all the services. This part of the Individual Service Provisioning set may be well suited to act as the user's front end to the range of Semantic Web services that are forthcoming.

Chapter 5

Summary of the Papers

The five papers contained in this thesis describe the work of Individual Service Provisioning in detail. They also indicate the chronology of the work, as they follow the development of the system from its inception to its current state.

When reading this chapter, and the introduction as a whole, keep in mind that the introductory text, on one hand, serves the purpose of describing the work as a whole, wherein each significant part is treated according to its importance as a contribution to the overall work. The papers, on the other hand, each have a very narrow focus dealing with a specific topic; therefore, as a result, the distribution of text over the collection of papers may not appear to accurately mirror the overall work.

For each paper, this chapter summarizes the work, describes the time frame and the project environment in which it took place, and explains the distribution of the work among the paper authors. The summaries are kept short intentionally since the papers themselves contain the pertinent details.

5.1 Paper A (Licentiate Thesis) sicsDAIS: Managing User Interaction with Multiple Agents

The first document, Paper A, is Fredrik Espinoza's Licentiate thesis, entitled "sicsDAIS: Managing User Interaction with Multiple Agents". It describes the very beginning of the work and the ideas that expressed the original vision. These ideas have evolved and matured throughout the work, but they were very definitely expressed at this early stage. As one example, many of the visions that were later developed, are described in the future

work section, at the end of the Licentiate thesis. The PhD thesis covers this work in Sect. 3.2.1.

The work described here was done within the context of the KIMSAC EU project. Fredrik Espinoza did the design, and almost all of the implementation work. Olle Olsson and Markus Bylund at SICS contributed with some discussion around the design and small parts of the implementation work. Fredrik Espinoza wrote the thesis.

The following three papers describe the three main components of Individual Service Provisioning: the personal service environment, the service creation tool, and the provisioning system.

5.2 Paper B sView - Personal Service Interaction

Many of the visions of sicsDAIS surface again in the second paper, Paper B, "sView - Personal Service Interaction", which describes the sView system, the platform that forms the personal service environment for Individual Service Provisioning. The paper describes the motivation for creating an open service platform, including critique of the World Wide Web, another specimen in this category. Moreover, the benefits of the platform, including network independence, ubiquitous access, scalability, and interactivity, are described. This paper corresponds to Sect. 3.2 in the thesis.

This paper represents the first documentation describing sView and the work it describes was done partly in the SITI funded project I3SVIEW, for which Fredrik Espinoza was the project manager. Fredrik Espinoza and Markus Bylund, the coauthor of the paper, created the design, which builds on the work from the Licentiate thesis (Paper A). In this paper, Fredrik Espinoza and Markus Bylund also share the work of the implementation. The paper was written jointly. The paper is published in *"Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology"* (PAAM 2000), Manchester, UK, April 2000.

5.3 Paper C ServiceDesigner: A Tool to Help End-Users Become Individual Service Providers

Paper C, "ServiceDesigner: A Tool to Help End-Users Become Individual Service Providers", describes the design and implementation of a tool that enables end users to create their own services. The ServiceDesigner allows users to combine Web Services into sView compatible services that can be executed in the personal briefcase. This paper corresponds to Sect. 3.3.

This paper describes work done in a sub-project of the I3SVIEW project. It was started in the fall of 2000 and is still ongoing (December 2002). The design of the ServiceDesigner was done by Fredrik Espinoza and the implementation was done by Ola Hamfors as his Master's work with Fredrik Espinoza as the SICS appointed supervisor. The paper was written by Fredrik Espinoza with parts of the text contributed by Ola Hamfors. It has been accepted for publication in the proceedings of " Hawaii International Conference on System Sciences" (HICSS-36), scheduled for January 6–9, 2003, on Big Island, Hawaii.

5.4 Paper D Generic Peer-to-Peer Support for a Personal Service Platform

Paper D, "Generic Peer-to-Peer Support for a Personal Service Platform," mainly describes the network connectivity part of service provisioning. This is based on the peer-to-peer model, a decentralized network architecture that fosters scalability, robustness, and openness. The Briefcase Connectivity system is a provider service in sView that allows other services to become peer-to-peer nodes. The system is used in Individual Service Provisioning to provide a convenient access layer for user created services. This paper corresponds to Sect. 3.4 in the thesis.

The described work was done within the context of the FEEL EU project for which Fredrik Espinoza is the SICS partner leader. This project is currently ongoing (December 2002). The design of the Briefcase Connectivity system was done by Fredrik Espinoza with detail level choices made by the coauthor Lucas Hinz. The implementation was done by Lucas Hinz, as part of his Master's work, with Fredrik Espinoza as the SICS appointed supervi-

sor. The paper was written by Fredrik Espinoza with sections contributed by Lucas Hinz. It has been accepted for publication in the proceedings of "The 2003 International Symposium on Applications and the Internet" (Saint 2003), Orlando, Florida, January 27–31, 2003.

5.5 Paper E Towards Individual Service Provisioning

The final paper, Paper E, "Towards Individual Service Provisioning", describes the central issues of Individual Service Provisioning and consolidates the vision and the ideas of the other papers.

The design of Individual Service Provisioning was done by Fredrik Espinoza and the paper was written by Fredrik Espinoza. The paper has been accepted for publication as a short paper at "2003 International Conference on Intelligent User Interfaces" (IUI 2003), scheduled for January 12–15, 2003, in Miami, Florida.

Chapter 6

Conclusions and Future Work

Any user can publish a web page—the concept is simple: you design your page in a visual editor, submit it to a publishing house (someone's web server), and then you survey your work in a web browser. This takes approximately 10 minutes. The individual provisioning of services should, and can be, just as simple. Our provisioning system is a combination of ServiceDesigner, Briefcase Connectivity—the peer-to-peer network system with SharedServicesLoader, and sView. Using these components, the act of creating, publishing, and using a service becomes "simple as web."

Individual Service Provisioning provides benefit to users: more specialized services, quicker access to specialized services, and less dependency on big software houses. The following section summarizes some of the lessons learned in this work.

6.1 Lessons Learned

Within the area of service platforms we have studied technologies, designed and built prototypes, and increased our understanding in the following areas:

Platform architectures. We have studied and learned about the demands of platforms for service delivery and access and for a personal user experience. We have built a mobile, personal, and highly integrated prototype system that implements our findings.

Ubiquitous Access. Services should be accessible on all kinds of devices. What support for this can and should a service platform provide? We have studied this problem and provided support for services to render their user interfaces on a variety of devices, for example over WAP, the Web, or using graphical user interfaces. We (the OASIS group)

have also worked on a more generic rendering system that dynamically creates the appropriate interfaces (Nylander and Bylund, 2002).

Web Services. New commercial initiatives along with a broad open-source effort aim to provide basic service building blocks over the network to developers of applications. We have studied Web Services and provided support for end user access to such services in a service platform. Our work includes full support for the web service protocols SOAP and WSDL, as well as support for interoperation between web services.

Service Interoperation. Getting disparate and provider independent services to interoperate is a big challenge. We have studied the problems and built on our knowledge of the service platform and Web Services to provide real-time support to end users for the creation of new services that interoperate inside the service platform.

Service delivery. To use services they must first be found, chosen, accessed, and finally rendered. We have studied and implemented support for varying technologies such as JINI, automatic service discovery, and dynamic loading of services. This support has been integrated into the service platform as pluggable components.

Security. When downloading services dynamically it is important to ensure a high degree of security for the platform itself as well as for the individual services. We have therefore studied these demands and provided support for such security in the underlying platform.

Peer-to-peer systems. We have studied existing peer-to-peer systems and applications and designed and implemented our own generic peer-to-peer system as support for the service platform. Our system enables a service provider to focus solely on the core functionality of the peer-to-peer based service while relying on our generic peer-to-peer support for the actual network connections. Our peer-to-peer implementation builds on what we see as a possible future standard system in this area, namely Project JXTA (Gong, 2001).

Within the field of ubiquitous computing, we have focused on the areas of:

Context dependent services. We have studied the demands of providing a user with a platform for accessing context dependent services such as services that are available in a specific location. In FEEL, the related EU-funded project, we have tested our platform and implementations on a project wide basis across all partners of the project.

Simulating context information. When developing context aware services we have found that it is often difficult to provide the actual context information throughout the development process. It may then be helpful to simulate context information. For example, we have built a context simulator tool called QuakeSim (Bylund and Espinoza, 2002) that simulates the user's location in a 3D environment and feeds this data to position aware services. We have successfully used QuakeSim to test and demonstrate our *GeoNotes* system, a system of virtual post-it like notes that can be placed in the real world (Espinoza et al., 2001).

Additionally, we have borrowed many of the ideas of *social computing* and built support into the platform for some of the aspects:

Community building. We have come to believe that service provisioning is a very social activity. It may help users tremendously to know or be aware of other users' activities in choosing, using, and understanding services. We have plans for supporting such social functions by creating specific pluggable modules for the platform that can provide for example ratings, advice, reviews, and statistics of usage.

Humans in the loop. We have studied service interoperability especially carefully and have come to the belief that much knowledge about the somewhat complicated task of creating interoperating services may be shared among users. In the trivial case, one user's created services can be made available to other users. In another case, the system may analyze the user's current set of services and from this and from knowledge about other similarly equipped users suggest other interesting or useful services.

We believe the new "service model" of computer usage calls for new technology for service platforms and ubiquitous computing and that social computing is a possibly interesting and useful design model.

The service model will become increasingly important in the near future: this is clearly indicated by the development of new mobile devices, increased interest in ubiquitous computing research, and brand name companies commitment to new technologies such as Web Services. The competence we have acquired throughout this project seems to be perfectly positioned for this new age of computing.

6.2 Future Work

The future work in Individual Service Provisioning will be focused around three concepts: *Loose Coupling Provisioning*, *Trusting Services*, and *Improving the ServiceDesigner*.

6.2.1 Loose Coupling Provisioning

We know that finding appropriate services can be hard for people; this problem is similar to that of finding the appropriate information on the Web. We also know that humans tend to be social in their day-to-day lives, making use of the opinions of other people, for example by reading reviews or by asking for directions. Building on our model of *loose couplings*, our solution to the problem of finding services will be based on social mechanisms.

The first purpose of loose couplings is to allow a user to connect together services that run in sView. This is similar to creating new services with ServiceDesigner but in this case, no new service is generated. Instead, useful couplings are made which move data between *sView services* in a structured, well-defined and automated way. For example, if the user has a service for keeping track of personal setting such as address, phone number, and credit-card numbers; and a shopping service, these may be connected using loose couplings. The user can define that when some action occurs in the shopping service, such as the placing of an order, some of the necessary data should be gathered from the preference service; loose couplings thus enables an individual user to automate data flows between services. In the following text, we describe possible future work concerning loose couplings.

A loose coupling defines the useful and appropriate connections between two or more services. It contains information about which services are involved, which functional components these services support, which couplings are in affect, the time and date when the coupling was created, a natural language description of the coupling (the purpose and the function of the coupling), and a description of the author of the coupling. The schema is stored as an XML coded text file.

First, we need to develop a tool to facilitate the usage of loose couplings. This *loose coupling manager* will keep track of the couplings and handle data flow between components. It should enable a user to create, edit, view, and administer couplings. This tool might overlay graphical cues on top of the services to emphasize their couplings. It should also allow users to manage access rights of services involved in loose couplings, for example by quickly isolating a service that shows signs of malevolent behavior. At any time, a user should be able to visualize which loose couplings are in effect and

each coupling's purpose (as described in the natural language section of the coupling schema).

Second, we need to enable *loose coupling provisioning*, i.e., the automatic provisioning of loose couplings schemas to other users. When a user creates a loose coupling between two or more services, the loose coupling schema is created and saved in the user's internal store of loose couplings. When another user creates a loose coupling, couplings made previously by other users should be accessible. This way the user may be able to accomplish the required coupling functionality without creating the coupling by himself.

The system should also be able to suggest to the user loose couplings that are possible given the services in the user's briefcase. The loose coupling service will continuously monitor the existing services in the user's briefcase and it will search the distributed repository for matching loose couplings. When such as match is found it will be presented to the user as a suggestion for improving the overall function of the services. The user may then accept the suggestion and try out the coupling. The loose coupling provisioning system will use Briefcase Connectivity to connect all users of loose coupling provisioning in a peer-to-peer network. Thus, as users create loose couplings the total number of loose couplings that the system knows about increases.

Third, we would like to add meta-level services such as rating, ranking, and recommendation services, to complement the loose coupling and ServiceDesigner systems. As long as a loose coupling is in use it is also stored in the user's loose coupling store. This means that a certain loose coupling may exist in many users' storage areas. This fact can be exploited to make more appropriate suggestions as to which loose couplings might be useful. As the loose coupling service continuously searches for matching loose couplings, it also keeps track of how many other users are currently using each coupling. A coupling that is used by more users gets a higher rating than one that is used by fewer users. When a match is found in the user's briefcase, the most popular coupling is suggested to the user.

However, the popularity of a coupling is not static. If a user finds that a coupling is not working, if it is malevolent, or if there is another coupling for the same purpose that does the job better, the user will delete the coupling. As the user deletes the coupling, it is removed from the user's store and its popularity rating is decreased.

6.2.2 Trusting Services

We are currently applying for funding to start a project on trust in Individual Service Provisioning. Trust is important when you have individuals providing services, since you no longer can depend on traditional trust mechanisms

such as brand names of well-known service or application providers, retailers, and distributors. The number of professionally created services will be great; the number of individually created services will be even greater. When the number of services increases, the risk is that it will be harder to ensure their quality and the user's security. Because of its peer-to-peer based provisioning system, the characteristics of Individual Service Provisioning include: a lack of centralized coordination, a lack of a central database, incomplete knowledge of the whole system, global emergence of behavior from local interactions, autonomous peers, unreliable peers, and connections between peers. From this follows the research questions (Aberer and Despotovic, 2001):

1. Which model of trust should be used? Should it be based on statistics of prior market based or socially based experiences of peers or on game theory?

2. What algorithms can be used to establish trust given the data above? One has to consider that the sources of data may not be trustworthy, or available.

3. Can the trust system be made to scale? The data collection and communication between nodes must scale to the number of nodes.

We need new tools and middle-ware that enables this trust despite the new models of service provisioning and delivery. The project will study other fields in which trust is an emergent effect of the social network, such as sociology and group dynamics, and synthesize possible trust mechanisms for the Individual Service Provisioning architecture. Then, in a later stage, these mechanisms will be implemented and tested within the framework. The author Neal Stephenson elaborates on this idea (Stephenson, 2001):

> "Basically I think that security measures of a purely technological nature, such as guns and crypto, are of real value, but that the great bulk of our security, at least in modern industrialized nations, derives from intangible factors having to do with the social fabric, which are poorly understood by just about everyone. If that is true, then those who wish to use the Internet as a tool for enhancing security, freedom, and other good things might wish to turn their efforts away from purely technical fixes and try to develop some understanding of just what the social fabric is, how it works, and how the Internet could enhance it. However, this may conflict with the (absolutely reasonable and understandable) desire for privacy."

The availability of more services is beneficial to users and providers alike, but with more services, the problems of trust and security increase, especially when each user can be a service provider.

6.2.3 Improving the ServiceDesigner

We foresee two possible avenues to improve the ServiceDesigner: adding a more powerful coupling mechanism, which includes semantic capabilities, and introducing collaborative development of services.

Introducing Semantics into ServiceDesigner

The connection mechanism of the current version of ServiceDesigner is rather simplistic; it allows the user to connect the data output of one functional component to one of the input channels of another. Although several components can be chained together in this way, to form an arbitrarily complex structure, there is little support for procedural constructs such as iteration, sequence, and conditional execution.[1] By adding these types of constructs, we can provide a more complete toolbox for creating services that are more sophisticated.

We can also add support for semantic understanding of couplings. This is an interesting point, since we do not intend to replace the user involvement with automatic semantic-based matching and coupling, but rather to complement it.

In the related work section, we examined some of the Semantic Web technologies. In the ServiceDesigner, these could be used to gather information about the user's need, find appropriate resources in the network, and assemble the resources to fit the need. The resulting service would fit nicely in sView and would possibly be beneficial to other users too. We intend to explore this possibility and position sView as a suitable environment for interaction with Semantic Web services. The more interesting approach, however, is that of combining the semantic-based matching with our human instruction.

In today's ServiceDesigner, the user manually picks services and functional components from resources such as web directories. This implies that the functional components used in a service composition are *exactly* specified, i.e., exactly those services that are picked will be used in the composition. This can be a frail solution, since there is no guarantee that the chosen components will actually function when the service is eventually used.

[1] There is, nonetheless, a timed repeater component available in ServiceDesigner.

A more robust solution would entail picking services and functional components from a set of available *concepts*. Since the Semantic Web technologies rely heavily on ontologies for declaring properties and capabilities of services, ServiceDesigner could use these same ontologies as the palette from which services and functional components are chosen. Then, when the actual service is created, it is not with concrete instances of functional components, but rather with functional component *types*; the process of resolving a type to an instance would take place at run-time when the coupling is executed. After a service has been created, it could be classified according to an ontology, in order for other users to better understand its purpose (similarly to how future versions of web design tools may have the ability to automatically tag pages with semantic tags (Hendler, 2001)). It could also be automatically tested and verified (Narayanan and McIlraith, 2002).

By leveraging the emerging Semantic Web technologies of semantic matching, ontologies, and run-time resolution, ServiceDesigner could produce even better results. Widely used ontologies may be used successfully for common concepts and services, such as travel booking and financial information, but for more unusual and niched services, there may not exist appropriate ontologies. In those cases, our basic solution, of handpicking the relevant services, will still fit the bill.

Introducing Collaborative Development

Every software system sold on the market today should include open source code to enable buyers to fairly judge the quality and value-for-money of the products they buy (Connell, 2002). This does not imply that the source code should have an open source license, but rather that software systems, like other engineering products such as bridges, houses, and airplanes, should be open for inspection. This could improve the quality of software, partly because it would be plain to see who is responsible for what, and partly because the responsible party could be made accountable for any deficiencies.

Along this vein, we intend to introduce collaborative development into ServiceDesigner. All services created with ServiceDesigner should be open for inspection, and more importantly, it should be possible for anyone to improve services. By enabling inspection of services, much of the security risks disappear as malevolent services can be discovered and announced as such. By permitting incremental improvements of services, we hope to attain a kind of evolutionary development process, in which good services are made better and bad services are made obsolete. This system would be reminiscent of open source development projects where many developers collaborate to produce fine quality software, although the “projects”, or services, would

not be hosted at a particular site, but would instead "float" in the network as virtual open source projects. The Individual Service Provisioning system would become a virtual open source development environment. To implement this system, we will need to add a formal model for couplings which can be expressed, inspected and modified; version information to keep track of the development progress and to allow users to choose the best or newest versions of services; and mechanisms by which developers gain credit (or are made accountable) for services they produce.

6.2.4 The True Future of Individual Service Provisioning

The true future of Individual Service Provisioning, is of course entertainment.

> "After people have the things they need to live, everything else is entertainment. Everything."

Madame Ping, in Neal Stephenson's *The Diamond Age* (Stephenson, 1996). ■

Bibliography

Aberer, K. and Despotovic, Z. (2001). Managing Trust in a Peer-2-Peer Information System, *in* H. Paques, L. Liu and D. Grossman (eds), *Proceedings of the Tenth International Conferenceon Information andKnowledgeManagement(CIKM01)*, ACM Press, New York, pp. 310–317.

Adar, E. and Huberman, B. A. (2000). Free Riding on Gnutella, Web. *http://www.firstmonday.dk/issues/issue5_10/adar/index.html

Ankolekar, A., Burstein, M., Hobbs, J. R., Lassila, O., Martin, D. L., McDermott, D., McIlraith, S. A., Narayanan, S., Paolucci, M., Payne, T. R. and Sycara, K. (2002). DAML-S: Web Service Description for the Semantic Web, *in* I. H. . J. Hendler (ed.), *The Semantic Web - ISWC 2002*, Lecture Notes in Computer Science 2342, Springer Verlag, pp. 348–363.

Ankolekar, A., Burstein, M., Hobbs, J. R., Lassila, O., Martin, D. L., McIlraith, S. A., Narayanan, S., Paolucci, M., Payne, T., Sycara, K. and Zeng, H. (2001). DAML-S: Semantic Markup for Web Services, *International Semantic Web Working Symposium (SWWS)*.

Apa (2002). Apache SOAP FAQ, Web. *http://xml.apache.org/soap/faq/faq_chawke.html

Bauer, M. and Dengler, D. (1999). InfoBeans - Configuration of Personalized Information Services, *Proceedings of the International Conference on Intelligent User Interfaces*, pp. 153–156.

Bennett, F., Richardson, T. and Harter, A. (1994). Teleporting - Making Applications Mobile, *Proceedings of 1994 Workshop on Mobile Computing Systems and Applications*, Santa Cruz.

Berners-Lee, T., Hendler, J. and Lassila, O. (2001). The Semantic Web, *Scientific American* **284**(5): 34–43.

Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B. and Kubiatowicz, J. (1999). OceanStore: An Extremely Wide-Area Storage System, *Technical Report UCB/CSD-00-1102*, University of California at Berkeley, Computer Science Division, Berkeley, California 94720.

Boman, M. (2000). *Implementing Services for a PSE – an Evalutation and Performance Analysis*, Master's thesis, Uppsala University.

Boman, M., Bylund, M., Espinoza, F., Danielson, M. and Lybäck, D. (2002). Trading Agents for Roaming Users, *Proceedings of the Tokyo Mobile Roundtable*, Tokyo. CD-rom.

Bylund, M. (2001). *Personal Service Environments - Openness and User Control in User-Service Interaction*, Licentiate of Philosophy Thesis, Uppsala University.

Bylund, M. and Espinoza, F. (2000). sView – Personalized Service Interaction, *in* J. Bradshaw and G. Arnold (eds), *Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 2000)*, The Practical Application Company Ltd., Manchester, UK, pp. 215–218.

Bylund, M. and Espinoza, F. (2002). Testing and Demonstrating Context-Aware Sservice with Quake III Arena, *Communications of the ACM* **45**(1): 46–48.

Bylund, M. and Wærn, A. (2001). Personal Service Environments - Openness and User Control in User-Service Interaction, *Technical Report T2001:07*, Swedish Institute of Computer Science, Kista, Sweden.

Charlton, P., Espinoza, F., Mamdani, E., Olsson, O., Pitt, J. and Somers, F. (1997). An Open Agent Architecture Supporting Multimedia Services on Public Information Kiosks, *Proceedings of PAAM 97: Practical Applications of Intelligent Agents & Multi-agent Systems*, pp. 445–466.

Charlton, P., Espinoza, F., Mamdani, E., Olsson, O., Pitt, J., Somers, F. and Wærn, A. (1997). Using an Asset Model for Integration of Agents and Multimedia to Provide an Open Service Architecture, *Proceedings of ECMAST97: Second European Conference on Multimedia Applications*, Springer-Verlag, pp. 635–650.

Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S. (2001). Web Services Description Language (WSDL) 1.1, Web. *http://www.w3.org/TR/wsdl

Coen, M. H. (1998). Design Principles for Intelligent Environments, *Intelligent Environments, Papers from the 1998 AAAI Spring Symposium*, AAAI Press, pp. 37–43. Technical Report number SS-98-92.

Coen, M. H. (1999). The Future Of Human-Computer Interaction or How I Learned to Stop Worrying and Love my Intelligent Room, *IEEE Intelligent Systems* **14**(2): 8–19.

Cohen, P., Cheyer, A., Wang, M. and Baeg, S. (1994). An Open Agent Architecture, *AAAI Spring Symposium*, pp. 1–8.

Connell, C. (2002). All Source Code Should Be Open, Web. *http://softwaredev.earthweb.com/sdopen/article/0,,12077_1452091,00.htm

Davies, N. and Gellersen, H.-W. (2002). Beyound Prototypes: Challenges in Deploying Ubiquitous Systems, *IEEE Pervasive Computing* **1**(1): 26–35.

Dey, A. K. (2000). *Providing Architectural Support for Building Context-Aware Applications*, PhD thesis, College of Computing, Georgia Institute of Technology.

E2 (2002). E2 Home, Web. *http://www.e2-home.com/

Ele (1999). Electrolux Screen Fridge, Web. *http://www.electrolux.com/screenfridge/

Espinoza, A. (2002). Conceptualizing User Interaction in a Multi-Service Environment, *Technical report*, Swedish Institute of Computer Science. T2002:19.

Espinoza, F. (1998). *sicsDAIS: Managing User Interaction with Multiple Agents*, Licentiate of Philosophy Thesis, Stockholm University.

Espinoza, F. (1999). sicsDAIS: A Multi-Agent Interaction System for the Internet, *Proceedings of WebNet 99–World Conference on the WWW and Internet*, pp. 1257–1258.

Espinoza, F. and Hamfors, O. (2003). ServiceDesigner: A Tool to Help End-Users Become Individual Service Providers, *Proceedings of HICSS-36, Hawaii International Conference on System Sciences*. Forthcoming.

Espinoza, F. and Hinz, L. (2003). Generic Peer-to-Peer Support for a Personal Service Platform, *Proceedings of Saint 2003*, Orlando, Florida. Forthcoming.

Espinoza, F., Persson, P., Sandin, A., Nyström, H., Cacciatore, E. and Bylund, M. (2001). GeoNotes: Social and Navigational Aspects of Location-Based Information Systems, *in* G. Abowd, B. Brumitt and S. Shafer (eds), *Proceedings of Ubicomp 2001: Ubiquitous Computing (LNCS 2201)*, Springer-Verlag, pp. 2–18.

Frank, M., Szekely, P., Neches, R., Yan, B. and Lopez, J. (2002). WebScripter: World-Wide Grass-roots Ontology Translation via Implicit End-User Alignment, *Proceedings of The Eleventh International World Wide Web Conference*.

Fre (2002). Freenet, Web.
*http://freenet.sourceforge.net

Garlan, D., Siewiorek, D., Smailagic, A. and Steenkiste, P. (2002). Project Aura: Toward Distraction-Free Pervasive Computing, *IEEE Pervasive Computing* **1**(2): 22–31.

Genesereth, M. R. and Ketchpel, S. P. (1995). Software Agents, *Communications of the ACM* **37**(7): 58–53.

Glance, N. S. and Huberman, B. A. (1994). The Dynamics of Social Dilemmas, *Scientific American* pp. 58–63.

Gnu (2002). Gnutella, Web.
*http://gnutella.wego.com

Gong, L. (2001). Project JXTA: A Technology Overview, Web.
*http://jxta.org/project/www/docs/TechOverview.pdf

Green, D. (1994). Emergent Behavior in Biological Systems, *Complexity International* **1**.
*Paper ID: green01, http://www.csu.edu.au/ci/vol01/green01/

Gruber, T. R. (1993). A Translation Approach to Portable Ontologies, *Knowledge Acquisition* **5**(2): 199–220.

Hamfors, O. (2001). *Service Designer - Lets the End-user Create her Own User-Interface to Web Services*, Master's thesis, Royal Institute of Technology, Stockholm, Sweden.

Hardin, G. (1968). The Tragedy of the Commons, *Science* (162): 1243–1248.

Heflin, J. and Hendler, J. (2001). A Portrait of the Semantic Web in Action, *IEEE Intelligent Systems* **16**(2): 54–59.

Hellman, K. and Hellman, S. (2002). *Applikationsintegration med XML*, Master's thesis, Department of Computer and System Science, Stockholm University/Royal Institute of Technology.

Hendler, J. (2001). Agents and the Semantic Web, *IEEE Intelligent Systems* **6**(2): 30–37.

Hendler, J. and McGuinness, D. L. (2000). Darpa Agent Markup Language, *IEEE Intelligent Systems* **15**(6): 72–73.

Hinz, L. (2002). *Peer-to-Peer Support in a Personal Service Environment*, Master's thesis, Uppsala University, Uppsala, Sweden.

ICQ (2002). ICQ, Web.
*http://www.icq.com

Katz, M. and Shapiro, C. (1994). Systems Competition and Network Effects, *Journal of Economic Perspectives* **8**(2): 93–115.

Kidd, C. D., Orr, R. J., Abowd, G. D., Atkeson, C. G., Essa, I. A., MacIntyre, B., Mynatt, E., Starner, T. E. and Newstetter, W. (1999). The Aware Home: A Living Laboratory for Ubiquitous Computing Research, *Proceedings of the Second International Workshop on Cooperative Buildings - CoBuild'99*.

Kohtake, N., Rekimoto, J. and Anzai, Y. (2001). InfoPoint: A Device that Provides a Uniform User Interface to Allow Appliances to Work Together Over a Network, *Personal and Ubiquitous Computing* **5**(4): 264–274.

Lassila, O. (2002). Serendipitous Interoperability, *in* E. Hyvnen (ed.), *The Semantic Web Kick-Off in Finland - Vision, Technologies, Research, and Applications*, HIIT Publications 2002-001, University of Helsinki.

Linthicum, D. S. (1999). *Enterprise Application Integration*, Addison Wesley Longman.

McDermott, D., Burstein, M. and Smith, D. (2001). Overcoming Ontology Mismatches in Transactions with Self-Describing Agents, pp. 285–302.

McIlraith, S. A., Son, T. C. and Zeng, H. (2001). Semantic Web Services, *IEEE Intelligent Systems* **16**(2): 46–53.

Moran, B., Cheyer, A., Julia, L., Martin, D. and Park, S. (1997). Multimodal User Interfaces in the Open Agent Architecture, *Proceedings of the 1997 International Conference on Intelligent User Interfaces (IUI97)*, Orlando, Florida, pp. 61–68.

Nap (2000). Napster, Web.
*http://www.napster.com

Narayanan, S. and McIlraith, S. A. (2002). Simulation, Verification and Automated Composition of Web Services, *Proceedings of the Eleventh International Conference on World Wide Web*, ACM Press, pp. 77–88.

Nwana, H. S. (1996). Software Agents: An Overview, *Knowledge Engineering Review* **11**(3): 1–40.

Nylander, S. and Bylund, M. (2002). Providing Device Independence to Mobile Services, *7th ERCIM workshop on User Interfaces for All*.

Paolucci, M., Kawamura, T., Payne, T. R. and Sycara, K. (2002). Matching of Web Services Capabilities, *in* I. H. . J. Hendler (ed.), *The Semantic Web - ISWC 2002*, Lecture Notes in Computer Science 2342, Springer Verlag.

Payne, T. R., Singh, R. and Sycara, K. (2002). Browsing Schedules - An Agent-Based Approach to Navigating the Semantic Web, *in* I. H. . J. Hendler (ed.), *The Semantic Web - ISWC 2002*, Lecture Notes in Computer Science 2342, Springer Verlag.

Rekimoto, J., Ullmer, B. and Oba, H. (2001). DataTiles: A Modular Platform for Mixed Physical and Graphical Interactions, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, New York, NY, USA, pp. 269–276.

Rouse, W. and Morris, N. (1986). On Looking Into the Black Box: Prospects and Limits in the Search for Mental Models, *Psychological Bulletin* **100**(3): 349–363.

Scheifler, R. W. and Gettys, J. (1992). *X Window System.*, Digital Press, Bedford, Massachussetts.

SET (2002). SETI@home, Web.
*http://setiathome.ssl.berkeley.edu/

Stephenson, N. (1996). *The Diamond Age*, The Penguin Group, London, England.

Stephenson, N. (2001). Summary of Talk Made at CFP 2000, Toronto, Web.
*http://www.well.com/user/neal/

Sun Microsystems, Inc. (1999). Jini Technology Architectural Overview, Web.
*http://wwws.sun.com/software/jini/whitepapers/architecture.pdf

Sun Microsystems, Inc. (2001). A Vision for Dynamic Java Services Provisioning, Web.
*http://wireless.java.sun.com/deploy/Java_Provision_WP_D4.pdf

Traversat, B., Abdelaziz, M., Duigou, M., Hugly, J.-C., Pouyoul, E. and Yeager, B. (2002). Project JXTA Virtual Network, *Technical report*, Sun Microsystems, Inc, Palo Alto, CA.

Ubi (1997). Ubiquitous Computing, Web.
*http://www.ubiq.com/hypertext/weiser/UbiHome.html

Waldspurger, C. A., Hogg, T., Huberman, B. A., Kephart, J. O. and Stornetta, W. S. (1992). Spawn: A Distributed Computational Economy, *IEEE Trans. on Software Engineering* **18**(2): 103–117.

Weiser, M. (1991). The Computer for the Twenty-First Century, *Scientific American* **265**(3): 94–104.

Weiser, M. (1993). Some Computer Science Problems in Ubiquitous Computing, *Communications of the ACM* **36**(7): 75–84.

Weiser, M. (1994). The World is not a Desktop, *Interactions* pp. 7–8.

Yamauchi, Y., Yokozawa, M., Shinohara, T. and Ishida, T. (2000). Collaboration with Lean Media: How Open-Source Software Succeeds, *Computer Supported Cooperative Work*, pp. 329–338.

Chapter 7

Paper A: sicsDAIS: Managing User Interaction with Multiple Agents

sicsDAIS: Managing User Interaction with Multiple Agents

Licentiate Thesis

October 1998

Fredrik Espinoza

Department of Computer and System Sciences
The Royal Institute of Technology and Stockholm University

Swedish Institute of Computer Science
Human-Computer Interaction and Language Engineering Laboratory
Box 1263, S-164 29 KISTA
Sweden
espinoza@sics.se

sicsDAIS: Managing User Interaction with Multiple Agents

Licentiate Thesis
by
Fredrik Espinoza

Abstract

User-computer interaction has changed in the history of computing; from batch systems to command line based systems and on to directly manipulated graphical systems. There is now a need for a new change, a need to incorporate *delegation*. Delegation gives users the option to offload tasks to software systems—agents—that perform the tasks for the user. This enables users to perform tasks that are difficult to perform using graphical user interfaces, tasks such as searching and retrieving data in large distributed networks or scheduled tasks that depend on future events.

In a near future, users will have to interact with multiple agents. The question is what this interaction will be like.

One possible form of interaction is through a common interface for all the agents. In such a system, users will access the agents' individual graphical user interfaces to receive information and describe and deploy tasks, while the interface application provides means for the agents to cooperate and coordinate their efforts by communicating and sharing data. Agents provide their own interfaces to SICS Dynamic Agent Interaction System (sicsDAIS)[1] as smaller versions of themselves, much as mobile agents, and sicsDAIS coordinates the presentations of these.

sicsDAIS is an example of a model of one interface for many agents. It is the central point where the user interacts with all agents, but it is not a pre-defined interaction, since agents can dynamically come and go, and the methods of interaction can change.

This approach is alternative to two other approaches. In the first, all agents provide their own disparate interfaces to the user. This makes coordination and

[1] SICS—Swedish Institute of Computer Science

sharing of data between agents difficult. In the second, there is one interface for all agents and all agents must conform to this interface without exception. This constrains agents in their expressiveness of the interface and it makes an open system difficult to achieve.

We will show that the first approach is the better choice from a system design perspective.

Contents

Acknowledgements

I would like to thank my supervisors Kristina Höök and Annika Waern at SICS, and Carl Gustav Jansson at The Royal Institute of Technology/Stockholm University. Special thanks to Kristina who found the thesis interesting summer reading.

As most of this work was done within the KIMSAC project, I would like to thank all partners in this project and especially Olle Olsson and Markus Bylund at SICS, Johan Bursjö and Simon Öhlmer previously at SICS, and the lads at Broadcom in Ireland. Olle and Markus have contributed with many ideas as well as implementation work. Markus especially, has during the later stages of the project worked closely with me in an effort to perfect the system.

I would also like to thank the people that have helped me with commenting the thesis: Nils Dahlbäck, Anna Jonsson, Anette Hulth, Fredrik Rutz, and Mark Tierney.

Thanks also to Adam Cheyer for being my external examiner.

This thesis is for my brothers and sister: Andreas, Nicolas, Erik, and Isabelle, and for my parents.

Chapter 1

Introduction

User-computer interaction has changed in the history of computing; from batch systems to command-line based systems and on to directly manipulated graphical systems. There is now a need for a new change, a need to incorporate delegation. Delegation gives users the option to offload tasks to software systems—agents—that perform the tasks for the user. This enables users to perform tasks that are difficult to perform using graphical user interfaces, tasks such as searching and retrieving data in large distributed networks or scheduled tasks that depend on future events.

In a near future, users will have to interact with multiple agents. The question is what this interaction will be like.

This thesis describes a user interface system called SICS Dynamic Agent Interaction System (sicsDAIS)[2], that facilitates a user's interaction with multiple agents. An agent in this context, is a software (or human) entity that provides some service that is accessible through a graphical user interface (GUI). In this system, a user is faced with, and is able to interact with, a combination of smaller graphical versions of the agents. These so-called content handlers[3] coexist and cooperate in sicsDAIS to enable the user-agent interaction.

The work described in this thesis is focused on making the access and coordination possible in one unified interface. It includes design and implementation of components in sicsDAIS that

[2] SICS—Swedish Institute of Computer Science

[3] Content handlers in sicsDAIS should more properly be called *interaction handlers*. The name content handler is used for historical reasons.

enable largely different agents to cooperate and present information to the user in the common sicsDAIS application.

This approach is alternative to two other approaches. In the first, all agents provide their own disparate interfaces to the user. This makes coordination and sharing of data between agents difficult. In the second, there is one interface for all agents and all agents must conform to this interface without exception. This constrains agents in their expressiveness and it makes an open system difficult to achieve. We will show in the thesis that the third approach is the better choice from a system design perspective.

1.1 Agents

As computation takes on new challenges in complexity and in terms of providing new services to users, a new methodology is evolving which provides the means to view or create systems with *agents*. By perceiving computational entities as agents, we can better understand the processes involved in complex computational systems and develop the systems accordingly.

The definition of an agent in the context of computer systems is imprecise. On one hand, Genesereth and Ketchpel define agents as software components that can *communicate* with each other using an "agent communication language" (ACL) [23]. Such a language should have agent-independent semantics (as opposed to messages in object-oriented systems that may vary in meaning between objects) and be able to express knowledge, as well as scripts, in a domain independent way. Systems may thus be created that promote interoperability between heterogeneous components to solve complex problems. Consequently, agents can be created independently of one another and yet function together by communicating using the ACL.

On the other hand, agents are defined as components that take on certain traits such as reactivity, autonomy, proactivity and collaborativity, to mention a few. The emphasis here is on the agents' fulfillment of the characteristics of the different traits. Some agents take on some of the characteristics and almost no agents take on all. The question is what makes an agent. This question will perhaps never get a definitive answer but a common approach to defining an agent is to state that an agent is a component that is viewed as an agent [71]. The rationale is that the main purpose of considering systems as agents is to clarify their function, boundaries, and relationships.

Agents may be single or composite software parts or they may even be human users that provide some service, but the common trait is that they serve some purpose and they do it without express intervention by the user (to some degree). The advances in software and hardware technology now allow us to perform tasks never before possible, tasks like bringing mobile computing with us on the road, or using the Internet as a global source of information. Agents can provide a way of conceptualizing (and realizing) the systems needed for the new tasks.

In the case of systems development, the advantage is much the same as that in using agent-based systems: when considering a complex computer software system as a collection of agents, the complexity of the system may be alleviated [13]. The abstraction is a way to simplification much as procedural, functional, and declarative programming as well as object-oriented ideas have been previously [42].

In this thesis, agents will be viewed in the context of their need to interact and share information with their users. This view is orthogonal to the definition of an agent, since it is feasible that any agent, regardless of type, may wish to interact with its user.

1.2 Interacting with agents

Some agents are specifically designed for interaction with users. These may be called interface agents or user-facing agents [50]. They may be realized as anthropomorphic characters or other shapes in the interface and they are distinguished by their main raison d'être of providing some service to the user in that interface. This can be services such as adaptive help, personal assistant services as for example in keeping track of appointments, or the service of aiding the user's memory while writing e-mail [16,64].

Other agents provide services in a multi agent system or in a distributed environment [51,14]. Their main purpose may be to cooperate with other agents to achieve a task, or to roam the network in search of information. We will focus on this type of agent in the thesis. We will show that sicsDAIS provides means for interaction with all types of agents, including interface agents and agents in multi agent systems, that may be implemented within its framework.

What happens when the user interacts with non-interface agents? How will the interaction take place? One can expect agent services to increase in number in the near future. Agents will be accessible not only on the personal desktop or from the local network, but also from sources on the Internet. This is in a way a recurrence of the situation when the first direct manipulation[4] interfaces, as desktop platforms for applications, became available to a broad audience (the PC and the Macintosh). The interface of the operating system allowed for a new kind of interaction between the user and the applications; interaction using windows to display information, and buttons and menus to manipulate the information. The applications implemented the functionality that was needed for the tasks in each application, using the new tools for interaction made available through the operating system. The designs of the new application interfaces were in many cases very different from one another. The problem was to allow the applications to present interfaces that were as efficient for the task at hand as possible while at the same time minimizing the differences between applications. This is where interface guidelines (for example [70,33]) appeared, to preserve uniformity in the functionality and appearance of the interfaces.

User-agent interaction now faces a similar situation. Agents are making services available to users in terms of a new mode of interaction—delegation. Delegation, in contrast to direct manipulation, allows users to delegate tasks to software agents that are responsible for completing the tasks [49,54]. Different agents have different responsibilities and agents can make use of one another to complete complex tasks. This new method of interaction with complex systems will perhaps at some point replace direct manipulation for some types of applications like route guidance [34] or personal assistants [21,46]. For most agent systems, however, a combination of delegation and direct manipulation will be needed.

Each agent will want to present itself in the best possible way, in terms of efficiency of use similarly to the case of applications. How will agents be made available to the user? How will the user cope with a multitude of different agents providing separate interfaces? One possible solution is to allow the agents to present themselves and carry on the interaction with the user through a common user interface which combines all the agents' interfaces but that still allows each agent its individuality. This thesis describes such a system—sicsDAIS—for user-agent interaction.

---

[4] An interface made up of buttons, menus and other interface components, which is manipulated by pointing and clicking with an input device.

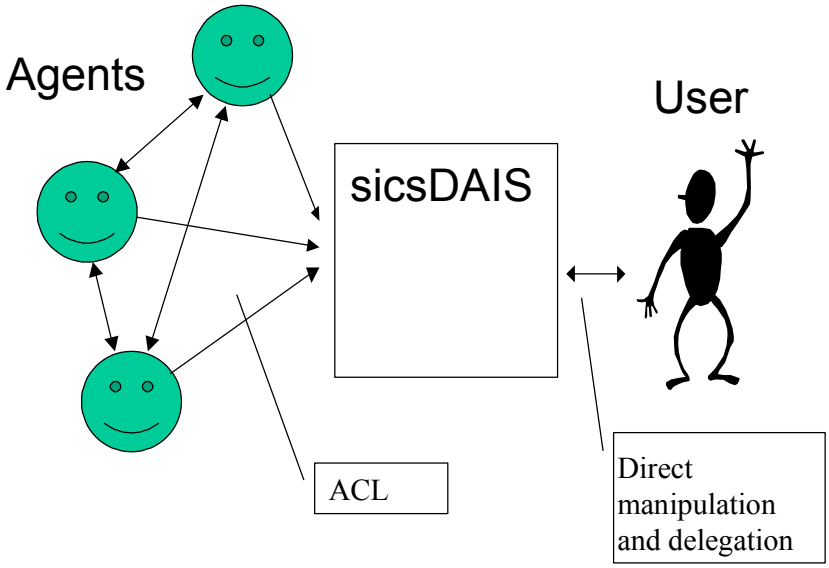

Figure 1. Agents communicating using an agent communication language (ACL); SICS Dynamic Agent Interaction System (sicsDAIS); and the user.

1.3 SICS Dynamic Agent Interaction System

sicsDAIS is a user interface and agent interaction system for interacting with multiple agents through a common interface (see Figure 1). The system is based on the idea that it is easier to provide for user interaction with multiple agents in a common forum than in disparate ones. Yet, each agent's individuality must not be lost to constraints on the agent's presentation to conform to rigid technical restrictions.

sicsDAIS is itself an agent. It communicates with other agents using a communication language and it performs reactively, proactively, and autonomously for its users (other agents and the human user). Mainly however, as the name implies, it is an interaction system. It receives requests from agents to present information to the user, and it dynamically creates these joint presentations/interaction sequences using *content handlers*, small pieces of interaction intelligence provided by the agents themselves. Content handlers are in a sense extensions of the agents, in the GUI. sicsDAIS provides the locale for the interaction including the layout system, communication between content handlers, and the interaction with the user.

User-agent interaction is a new way to interact with complex systems. The complexity of the systems suggests the change of interaction model from direct manipulation to delegation-manipulation (delegation with elements of direct manipulation to configure agents or access agents' data). To compare the evolution of delegation-manipulation to the evolution of direct manipulated interfaces one can align an interaction system such as sicsDAIS with the windowing systems of direct manipulation interfaces. sicsDAIS allows agents to exist near the user and it provides the building blocks for creating and conveying the interactions, but without placing constraints on the interactions through guidelines. It does however allow guidelines to be implemented and tested in the common agent interface.

1.4 Thesis outline

This thesis describes SICS Dynamic Agent Interaction System (sicsDAIS) and the rationale for its design.

The work on sicsDAIS was done within the frame (and as part of the system) of the ACTS project KIMSAC[5], further described in chapter 5. A three-year EU project (1995-1998), KIMSAC aims to provide integrated, kiosk-based access to government information services and intelligent and adaptive help to kiosk-users. Partners include: Swedish Institute of Computer Science, Sweden; Broadcom Éireann Research Ltd., Ireland; CAP SESA Telecom, France; CSELT, Italy; Digital Equipment Ireland Ltd., Ireland; Foras Aiseanna Saothair, Ireland; Teltec Ireland, Ireland; Social Welfare Services, Ireland; Trinity College, University of Dublin, Ireland; and Imperial College, U.K.

Parts of the work concerning the design of sicsDAIS have previously been published at Agents97 [9], PAAM97 [10], and ECMAST97 [11].

The thesis is organized as follows:

- *Chapter 2* provides an overview of other agent systems and their methods of presentation and interaction.
- *Chapter 3* describes the philosophy and the design of sicsDAIS.
- *Chapter 4* discusses related work.
- *Chapter 5* deals with sicsDAIS as a component in the agent-based system KIMSAC.
- *Chapter 6* contains a summary and a glance ahead to future work.

For readers familiar with software agents and presentation systems, chapter 3 of the thesis is the most relevant for describing the main ideas of this work. Chapter 5 may be interesting as a case-example of implementing and using sicsDAIS in a real project.

[5] KIMSAC: Kiosk-based Integrated Multimedia Service Access for Citizens

Chapter 2

Background

Let us start by reviewing a background of the field, divided into three parts. We will examine these topics because of their specific significance to the development of sicsDAIS.

In the first section, named *presentation systems*, we will examine systems for user presentation including the nearly static World Wide Web [4]. The web is static in the sense that most information available is prerecorded and statically made available as text pages (although there are examples of dynamic web pages that are generated at the time of presentation [15]). In this section we will focus on the way information is conveyed and presented to the user and concerning the web the way the information is described as to content and not to the exact mode of presentation.

We will continue with *plug-in architectures* including the Netscape [59] plug-in system [60] for adding capabilities to a web browser and the JavaBeans™[6] component architecture [39] for building visual applications (we will speak of the Java™ language [24] more generally in a later section). The focus here is on the idea of providing a plug-in architecture for complementing the functionality of a system (or defining the functionality of the system as in the case of JavaBeans). From the discussion, we derive a number of important ideas that we wish to remember, as we move from the classical direct manipulated interfaces to those based on delegation/direct manipulation. We also observe a number of implementation details.

In the last section, we introduce the concept of *agents* and the network where many agents exist. We then explore how the interaction between user and agent system might transpire.

[6] Sun Microsystems Inc.

Finally, we conclude the background section with a focus on a number of specific demands that may be placed on agents and multi-agent systems in terms of interaction with users.

2.1 Presentation systems

We wish to highlight the technologies used by the following systems when making information available, without being overly concerned with specific interface design issues that need to be taken into account (although these are certainly important). We will focus on methods that are available in making the presentations happen.

Presentation systems make information accessible to the user. They range from completely static systems where the presentations are pre-scripted in advance (PowerPoint [53], Macromedia Director [48]), to dynamic systems in which the coordination of the presentation is done during the presentation.

These systems may or may not contain interactive elements in terms of simple choices for proceeding in the presentation or choosing tracks, but the focus in not on interactivity. For example, in Personalized Plan-Based Presenter (PPP) [2], a system developed by the Artificial Intelligence (AI) research center at DFKI, the presentation is dynamically assembled at the time of presentation, taking into account temporal and other constraints. However, the user interaction is at a minimum.

Another system for dynamically constructing presentations is Coordinated Multimedia Explanation Testbed (COMET) [17], which generates and presents multimedia explanations. In this system, the presentation is built in a number of steps. First, the *content handler* component produces the full content for the explanation by accessing three databases of domain information and rules for constructing explanations. These include a static database of domain objects and actions, a diagnostic rule base, and a geometric knowledge base for the rendering of graphics.

The content is then passed to a *media coordinator* which annotates the content with information specifying the type of rendering (text or graphics) that is needed for each part. Each part is then generated. Lastly, the *media layout component* formats the content for the *rendering and typesetting software*. In this system, there is some measure of interactivity involved. The explanations that are provided by the system are shown at the request of the user. If an explanation involves a number of steps the user is required to move thorough the steps. The user can also browse the explanation steps in any order.

As we will see in later sections that describe sicsDAIS, the above systems are quite different from sicsDAIS although they all present information. sicsDAIS is in contrast a system that is focused on the interaction between the user and the system. sicsDAIS is also, as we will see, a system for making the *services* of agents available to the user.

In the next section, we examine the World Wide Web, an information system in which information is structured in a way that allows various tools to be used for access. The specifics of the presentations on the web are not explicitly specified and this allows for a lax interpretation as well as for a great deal of flexibility on the part of the viewing tools.

2.1.1 The World Wide Web

The World Wide Web is an unstructured collection of information that is made available through the interconnection of host computers all over the world. It is based on an Internet protocol called Hypertext Transport Protocol (HTTP) [32] and an accompanying markup

language for structuring information called HyperText Markup Language (HTML) [31]. On the server side, information is published using web servers that group related information or information belonging to some institution or individual. Clients access the information using special *web browsers*, programs that retrieve the information from the web servers and interpret the HTML coding of the web pages. The web is a continually evolving world of information where new services such as online encyclopedias and electronic marketplaces are appearing.

A web page is a collection of texts, pictures and other elements that can be viewed in a browser. The formatting of the information is in the form of tags that specify the general characteristics of text and the placing of the elements of the page. The tags describe such things as the relative size of a header or the alignment of a picture in regard to the surrounding text. Historically, the HTML language has been focused towards an unspecific type of formatting, i.e.; the format describes relative text size but not the specific font, or the approximate placement of an image but not the exact pixel coordinates. The reason for this is to enable a multitude of different types of browsers to access the information. Full-fledged browsers can handle all types of media but text-only browsers have no means for displaying graphics.

The web browser interprets the HTML information and renders a display that may vary slightly between browsers. The point is that it is up to the browser to decide how the information should be displayed. The author of the document may have had one thing in mind, while the constraints of the browser may result in a different presentation. The strength of the HTML formatting language is that it allows a gradual increase in the complexity of the presentation to the user depending on the tools that the user has available.

This is the interesting concept (for the thesis) regarding the web. The browser has the final say in the layout and rendering of the web page, but if the size of the browser window is changed, the presentation is changed accordingly and it is reformatted. The second point to be noted here is the need for dynamically built presentations as opposed to the mostly static presentations of the simple presentation systems and the web. In interacting with an open community of agents, the interaction and presentation screens must be dynamic to allow for new services and functions to be introduced.

2.2 Plug-in architectures

Netscape plug-ins is an architecture for extending web browsers' abilities to render new kinds of information. The essential issue is the fact that a browser can be equipped with plug-ins that add to the set of functions it can provide in terms of presenting information. This is similar to the scheme used in sicsDAIS for adding capabilities using content handlers as a means for providing interaction with agents (as we will see in section 3.5).

The JavaBeans component architecture for building visual applications, allows an application builder to assemble ready-made components into an application. The components are often general to be easily modified to suit the needs of the application. This system is similar to sicsDAIS in that a complete application is built from general components but with the important difference that in sicsDAIS presentations and interaction sequences are built dynamically during run-time.

2.2.1 Plug-ins

As we mentioned above, a web page may contain text and graphics, but also other types of objects, such as plug-ins. A plug-in is a component that is specially built to handle a specific type of data, such as audio or video. Along with text or graphics, a web page author may wish to include some type of data that the majority of browsers are incapable of rendering. The author marks the specific data with a tag that specifies which plug-in that is required to render this data. As the web page is loaded, the user's browser will also load the required plug-in as appropriate. Browsers capabilities can thus be expanded by installing plug-ins for new types of data.

The term plug-in stems from the manner in which the component is incorporated into a browser's set of abilities. It is installed on the user's machine along side of the browser and is called upon to plug into the browser when objects of the specified type are encountered. As the web page is rendered in the browser, the object being handled by the plug-in is allocated space in the page. This space is completely occupied by the plug-in and it is as if a separate program is running inside.

There are plug-ins for a variety of data types, such as PDF[7] files, VRML[8] files, and streaming audio.

Plug-ins are useful for extending the capabilities of a browser but there are less favorable properties of plug-ins as well:

- They are platform dependent. This means that a plug-in provider must develop specific versions of a plug-in for each deployment platform.
- They are installed on the client computer. This means that each user must download and install the plug-in before use. This also means that when the format of the plug-in changes, a new version of the plug-in must be created, downloaded, and installed.

Despite these minor deficiencies, plug-ins serve to improve the usefulness of browsers a great deal. This concept is central to the design of sicsDAIS, as we will see in section 3.

2.2.2 JavaBeans

JavaBeans is an architecture for building and using visual components. Using the JavaBeans model, software creators can define Beans, small visual components that can be used stand-alone or in combination, that may be used by application builders to compose applications. The model describes the individual Beans and the interdependencies of Beans, which allows for building and using application builders to combine JavaBeans into applications. For example, smaller Beans such as buttons or lists can be combined in a visual building tool to create a complete application. In the tool, the parameters of the Beans are set along with the events that may be caused by the Beans and the reactions that those events should cause.

The JavaBeans model is platform independent. Beans may be nested in each other and will then be able to access the full spectrum of functionality that is available to a Bean. Beans that are incorporated into some platform specific document or application such as a web browser will have to conform to the platform specific component architecture. The JavaBeans application programming interface (API) is designed to transition smoothly across platforms

---

[7] Portable Document Format, [1]

[8] Virtual Reality Modeling Language, [72]

so that a certain function that is not available on a platform is simulated or achieved in an alternative way. This leaves Beans developers one API that will function across all platforms.

"A JavaBean is a reusable software component that can be manipulated visually in a builder tool." [39]. It is a piece of code that describes some functionality in terms of both the visual interface and the underlying functions. Many JavaBeans have the visual presentation aspect as the main characteristics, while there are Beans that have no interface at all. A builder tool is a tool for tailoring the Beans to the particulars of a certain application as well as for combining different Beans into the application. Examples of tools are web page construction tools, visual application builders, or GUI builders. Some tools are completely visually oriented while others may provide a scriptable interface for controlling the individual Beans.

Individual JavaBeans provide certain common functionality:

- They support events. Events allow Beans to communicate in a JavaBeans application.
- They allow for reflection and introspection, which means that a builder tool can establish, by looking at a Bean, the set of operations that it handles.
- They allow for customization of properties that govern the characteristics of the Bean.
- They support persistence. After being created, a Bean may be serialized for storage, while in this process storing an initial state that will be reinstated when the Bean is reactivated.

Events

Let us examine events more carefully. The JavaBeans events mechanism provides an architecture for connecting Beans together in an application. In this architecture, Beans act as event sources, notifying the surrounding environment (the building tool or the web browser, for example) or other components of state changes in the Beans. The model relies heavily on the Java model of events, based on the concept of event sources and event listeners. In this model, interested components, event listeners, register their interest in particular events with the event sources. When an event occurs, the registered event listeners are notified through method calls. The design of the JavaBeans event architecture allows the events that a Bean may fire (or listen to) to be discovered by other components.

Introspection

During the development of an application in a visual builder or during the execution of an application, the builder or environment needs to access properties, events, and methods of JavaBeans. To enable this access the environment must discover the methods, properties, or events in question and this is done through introspection. Introspection is a process in which the internals of a JavaBean are revealed to the enveloping environment. There is no specification language for this but rather a solution based on the reflection API of Java, which allows a Java object to be inspected to find out about its member variables and methods. A JavaBean should be able to reveal its innards without extra coding. However, if the creator wishes to provide more advanced means for the process, a special `BeanInfo` class may be provided (a table of parameters or any other presentation of the parameters that the creator wishes to use). The simple introspection is based on Java's reflection mechanisms [41], in which methods' profiles and attributes of a Java class are accessible programmatically. The reflection access to JavaBeans is built into the `Introspection` class, which also contains simple design patterns that help to interpret the names of the methods and properties.

Design patterns are the set of conventional names that are used for methods. For example, `getSomeProperty` to retrieve a property, or `setSomeOtherProp` to set the property. When a Bean author uses names like these, the process of discovery can be fully automatic. It is however, voluntary to conform to the design patterns, but the requirement is then to provide the `BeanInfo` class.

Properties

Properties of a JavaBean are attributes of the Bean that affect its appearance or its functionality in some way. They are internal data variables that may be accessed programmatically using accessor methods on the owner object. Let us say that a button has a label property called label. This property can be changed by calling the `setLabel` method or in a script by simply setting the property to the new value. The names of the accessor methods can be arbitrary, but standard naming conventions may be used (as mentioned above).

As a Bean is created, some or all properties of a Bean are usually set to default values. This means that the Bean will have a well-defined appearance or state when it is instantiated, regardless if the properties are set by the application builder. Properties can also be manipulated using a property sheet, an editable table used in visual application builders.

Customization

Customization is the process by which developers can modify the appearance and other properties of JavaBeans as they are assembled into an application. This way a JavaBean may be used in a variety of ways. Note that the Bean may be customized to fit a particular application, as it is being developed, not during runtime as an application is being used. As we will see in the discussion about sicsDAIS' content handlers (section 3.5), these are also customizable. The difference between the two is that content handlers may be customized during run-time. This allows for on-the-fly tailoring to specific users, sharing of content handlers between agents during the running of an agent-based application, dynamic reuse, among other things.

Properties are accessible in two ways, using the introspection method described above, or using a special `Customizer` class that may accompany the Bean. If the Bean is simple, the Bean developer needs to do nothing to allow customization. If the Bean is more complex, or if the Bean developer wishes to provide for example a wizard for customization, the code for this is stored in the customization class. A Bean that has been customized is stored by the application builder tool using the methods for persistence that are available. Then, when the application is run, the Beans are loaded from persistence with the customized state.

We conclude this section about plug-ins by mentioning the important ideas that we wish to remember in the design of sicsDAIS:

- In the context of this thesis, the main contribution of plug-ins is the idea of having specialized components that handle specific types of data. This corresponds in sicsDAIS to specialized content handlers for handling different data, different types of interaction, or different agents.
- The JavaBeans architecture allows interdependencies between Beans, communication between Beans using events, reflection of Beans for establishing the nature of a Bean, and customization to specialize Beans. These ideas are all included in the design of sicsDAIS.

- The JavaBeans system allows customization of Beans to suit various situations. sicsDAIS goes one step further, to allow content handlers to be modified and customized during run-time.

2.3 Agent systems

Now that we have examined a number of important properties of presentation systems and plug-in architechtures, we turn again to agent systems.

As discussed in the introduction, the term agent is used both to denote software components that can communicate, as well as software components that take on particular characteristics such as autonomy, reactivity, proactivity, and so on. We will now further investigate the realm of software agents. The following sections will discuss the nature of an agent, the role of the network, user interaction with agent systems, and finally multi-agent systems and agent architectures.

2.3.1 What is a software agent?

To be able to examine applications of agents and the user-agent interaction we must start by examining the agent itself. The concept of agents is well known from the AI community since the 1970s when Hewitt [29] defined the term "actor" as being:

> "a computational agent which has a mail address and a behavior. Actors communicate by message-passing and carry out their actions concurrently"

This is similar to a later definition of a software agent as being:

> "a software entity which functions continuously and autonomously in a particular environment, often inhabited by other agents and processes" [69].

There is no real definition of *agent* in the domain of software systems, but the term *software agent* seems to suggests an entity embodied in software that exhibits some of the characteristics of an agent as used in everyday speech. In the introduction of this thesis, we mentioned a number of characteristics that an agent can exhibit. This non-exhaustive list includes

- Autonomy—the agent acts on its own accord and in keeping with its knowledge and choices without human intervention.
- Reactivity—the agent reacts to stimuli from the environment.
- Proactivity—the agent takes initiative to act on its own and it may act without the express direction of the user or because of stimulus from the environment.
- Collaboration—the agent is able to collaborate with other agents.

The concept of agenthood (being an agent) may be discussed in terms of the following:

- An agent is intelligent, i.e., it performs in a manner typical of humans in terms of making choices, inferring facts, and keeping knowledge as a basis for the previous two.
- An agent does something for its user.
- Agents are useful with or without direct manipulation (more on this in section 2.3.4).

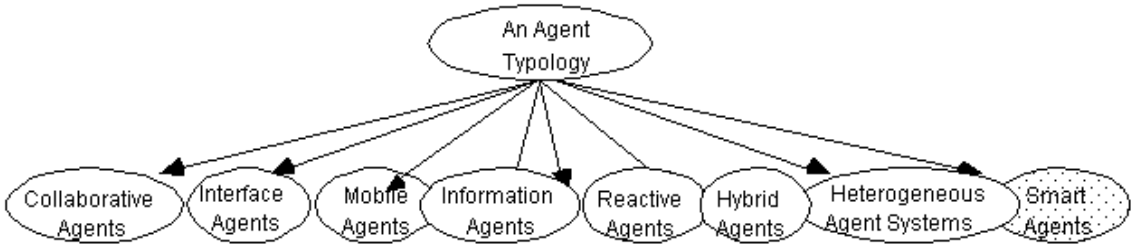

Figure 2. Nwana's topology of agents

The point of the discussion has been to distinguish software agents from other software components or even other man-made artifacts. The question has been why an agent is different from an expert system or a database, or even a thermostat. One answer is that a software agent is a software entity that adheres to some of the qualifying factors—autonomy, collaboration, etc—which have been mentioned previously. However, while one may very well be able to ascript the term *agent* to some entity, a thermostat for example (it acts autonomously, it provides a service to a user, it has an internal state, and it has the goal to signal the transition from one state to another), it may not always be beneficial to do so [68]. The complexity of the mechanism is in this case is not great enough to warrant the agent label.

An agent may be classified according to different criteria such as the task that the agent performs, qualities of the agent (like autonomy, reactivity, etc., as mentioned above), if the agent interacts with the user, and so on. When classifying agents in terms of the agents' degree of interaction with users one finds a number of different categories. Some agent systems never interact with any users. An example of such a system of agents could be agents that perform load balancing in networks [75] without any human intervention. Agents that monitor and filter a user's incoming e-mail messages have some degree of interaction with the user, perhaps at the time of setting up the service and at times of changing parameters [46]. Most of the time however, the agent performs a service that is noticeable only through the interface of the application the agent is controlling. This type of agent system is somewhere between the non-interactive systems and interface agents on a scale of user interactivity. Interface agents are a subset of user-interaction agents that use an anthropomorphic presence in the interface. The purpose of such an agent may be to assist the user in performing tasks in the otherwise directly manipulated interface. An example of this is the help assistant in the Microsoft Office

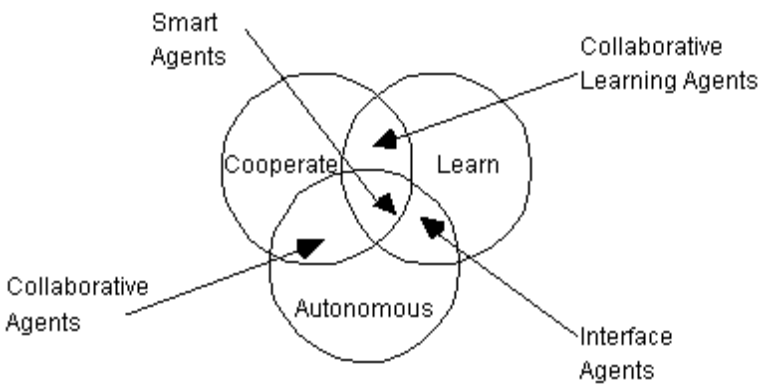

Figure 3. Nwana's attribute map of agents

suite [30].

In the context of this thesis the definition of software agent will not be completely specified, but the following idea seems to suffice; a software component that acts autonomously, pro- and reactively, with an internal state and a set of goals to achieve by interacting with its environment including its user(s). This definition places agents in the category of information agents in the topology of agents according to Nwana [61], and somewhere in the smart agent section in the attribute map (See Figure 2 and Figure 3). The agents that have been tested with sicsDAIS have been information agents with interfaces for performing problem solving and data manipulation. We will discuss these agents further in section 5.

Note that sicsDAIS is independent of the type of agents that manifest themselves in it.

2.3.2 Open networks

The concept of the network is central to the functioning of some agents. As we saw in the previous section, agents in themselves are usable, for example in interface applications such as the Microsoft Office Assistant, but their utility increases tremendously if they are allowed to cooperate or even migrate across a network. In a network, agents are able to combine forces to solve complex problems, utilizing distributed services, and the inherent parallelism of concurrent execution. The distributed nature of the network also allows agents to tackle problems that require an open and distributed system. Market interaction [27], service provision (like KIMSAC, section 5) and cooperative work, are examples of this.

The strength of a distributed agent-based system lies in the effects of combining many smaller parts into one system. The system becomes robust when there are many instances of the same functionality in case some become unavailable because of failure of some host or some part of the network. The system will be scalable if it is possible to involve more agents over a greater number of computers when the need arises. The system will also be open if it is possible to introduce new unforeseen capabilities without regard for the existing components.

KQML

In the introduction, we touched briefly on the matter of communication between agents. This is clearly an important topic in discussing the network as a platform for agents. We mentioned agent communication languages and Knowledge Query and Manipulation Language (KQML) [18,19] is one such language.

KQML is a language for communication between agents. The language is built in levels where the first level is the actual content of the message described in an arbitrary content language. The content is wrapped inside the KQML message, which is the second level. The communication language describes the purpose of the message as a performative followed by a list of arguments as key-value pairs. These may be the sender and receiver and other optional descriptors such as the content language etc. It is possible to add other useful information in the set of parameters and this allows for inspection of the message content despite its inaccessibility.

The performative describes the purpose of the message as a speech act [67]. The set of performatives forms the core of the language. The performative signifies that the content is an assertion, a query, a command, or some other mutually agreed upon speech act. It also describes the way the sender would like to have any reply delivered. The following is an example of a simple KQML message:

```
(ask-one
:content preferredSize();
:language JAVA
:ontology internal-state
:reply-with mess-32
:sender agent1
:receiver sicsdais)
```

This message requests the receiver to produce one answer to the question contained in the content part. The receiver should reply with the specified tag “mess-32” to identify the return message.

2.3.3 Mobile agents

A mobile agent is able to move between hosts in a network to interact with services in the different locations. It may collect and collate information from the different locations, and bring it back to the user. The benefits of using mobile agents are mostly non-functional as for most cases stationary agents can be built that perform the same tasks, but there are advantages in terms of cost. It may be more efficient to distribute the computation or to collect only the necessary information using mobile agents [28]:

- The network load may be decreased by allowing the agents to move to the source of the data instead of bringing the data to the client.
- Network latency may be decreased due to local computing of the mobile agents at the source of the information.
- Systems based on mobile agents may be more robust and fault tolerant than other systems because the agents can be duplicated in case some fail.

The Aglet framework [47] is an example of a mobile agent platform. This system may be used to build a network of mobile agent enabled sites (sites that the mobile agents may visit), as well as the mobile agents, Aglets, themselves. We will examine Aglets more carefully in section 4.2. Other mobile agent platforms include ObjectSpace Voyager [62], based on an object request broker (ORB) and Concordia [56], a framework also based on Java, for developing mobile agent applications. The three platforms mentioned here all attempt to provide an architecture for developing, deploying, and maintaining mobile agent applications. The Voyager system is somewhat more elaborate (in version 1.0) than the others in regard to remote messaging, the life span of agents, and mobility.

The anatomy of a mobile agent

A mobile agent needs a number of components [47]:

- A *state*. The state is frozen when the agent is moved and thawed as the agent reappears in a new location.
- An *implementation*. This is the code of the agent, which specifies the functionality of the agent.
- *Interfaces*. The agent needs interfaces for communication with the services it encounters as well as for communication with its user.
- An *identifier*. The agent needs a unique identifier to distinguish it from others, as well as for allowing the agent to be recognized and located.

- *Principals*. The agent needs principals for determining legal and moral responsibility.

The agent also needs a place to exist. This is the environment where the agent executes and this environment must be present on each host that the agent visits.

Network paradigms

To further explain the usefulness of mobile agents we will examine three network paradigms that are used for providing services to users in a network environment. These are (1) the *client-server paradigm*, (2) the *code-on-demand* paradigm, and (3) the *mobile-agent* paradigm.

1. The *client-server paradigm* is based on the server and the client. The server is the base of the service provided, it may be a database or processing engine, and it is located on a host machine on the network. The server provides the service to users by allowing users to connect on a session or transaction basis. The reason for keeping the server in one location is that the data or service provided is usually to large or computing intensive to be transferred to each client. Another advantage is that the service may be kept up-to-date in just one location, automatically resulting in fresh information for all users. The client in the client-server paradigm is any user or user-device that accesses the server. The client is usually a relatively lightweight component, which is easily transferred and installed at remote locations. The client accesses the server by querying it for the service, be it a database query or a request for some calculation. The results of the processing are retrieved by the client and presented to the user at the client location. The know-how of interacting with the server is on the server side.
2. In the case of *code-on-demand*, the set-up is similar to that of the client-server. A server provides the service in a remote location. Clients access the server to gather information or process queries. The difference is in where the know-how of doing the actual querying is kept. In the case of the client-server, it is kept with the server. In this case, it is also kept with the server but preceding the transaction, it is transferred to the client. The client is thus given the knowledge of how to perform the interaction with the server and the interactions can be more advanced. The know-how is kept with the server and transferred to each client.
3. In the final case, that of *mobile agents*, the agent contains the know-how of interacting with the server. The client is a mobile agent and it moves around the network accessing services. The agent may gather data from many services before returning to its place of origin to present the results.

2.3.4 Interacting with agents

The idea of viewing components in a distributed computational environment as agents is not new. Kay and Negroponte [43,58] envisioned the agent metaphor and the abstraction to a higher level that it brings and that is needed. In the case of humans interacting with computer systems, the same progress towards abstraction has been made. The move from command based interfaces to direct manipulation in interfaces [65] was needed because systems were becoming too complex and too awkward for people to use. Direct manipulation interfaces combined with icons representing objects (Windows, Icons, Menus, and Pointer (WIMP)) relieve much of the strain caused by the earlier command based systems. With the addition of metaphors for coupling tasks and objects to common everyday situations [65,7], the direct manipulation interfaces achieve an abstraction of sorts. The user does not have to know or

care about underlying processes of the operating system, but instead focuses on the method at hand using the tools provided in the interface.

Direct manipulation works well in many circumstances when the objects to be handled in the pursuit of the tasks can be represented on the screen and the tasks to be performed can be performed by manipulating the symbols. This is all right for a limited number of objects and tasks that are performed in a number of simple steps in sequence. However, when the numbers grow and the tasks involve delayed actions or complex combinations of simpler tasks, direct manipulation starts to fail. The alternative is to use the agent metaphor for these situations. Imagine a task like putting together a summary of all correspondences with a certain person during the month of May, or setting up a reminder for the times when correspondence like that in May arrives the next time. Tasks of this type are simpler to envision as delegations to software agents, than as direct manipulation actions [44].

User interfaces for direct manipulation applications are created to suit the tasks that will be performed in the applications. Each application has its own GUI that is designed to optimize the efficiency of the interaction between the user and the system. A GUI for a word processor application needs to display the text and controls for editing and formatting the text while a GUI for an interactive 3D world displays the world and navigational controls. Each application has a GUI that suits the needs of the interaction process of the specific application. The advantage of this is that each application will have an efficient interface (hopefully) but the disadvantage is that the interfaces will most likely differ. Interface design guidelines may be of help in the design of interfaces that are easy to learn to use because they look and function alike. However, the purpose of each application is of course inherently different so the user still has to learn to use the interface for each application.

The interaction between the user and agents will be both similar and different to this. It will be similar in that there will be numerous sources of agents and services available to the user much like there are many applications available to the user on the desktop. The services provided will be quite different from one another in terms of

- services provided (ranging from information retrieval to home appliance control)
- interface needs of the service (graphical views, text input, audio output)
- the interface platform (desktop, mobile telephone, appliance panel)

However, it will nevertheless be desirable to strive for commonality using guidelines in the interfaces of the service agents.

The difference lies in the fact that it is difficult to enforce a commonality between the services since they are not tied to the confines of a desktop, but rather distributed over a great range of platforms and appliances. The means for providing access to the services should therefore focus on providing facilities for issues such as accessibility, communication, sharing of data and the users personal profile, and collaboration using service contracts [73]. The means we speak of here is a framework for combining the interfaces of agent services and in this process enriching the user's interaction experience through the use of the facilities provided by the framework. An example is the Open Agent Architecture (OAA) [12], a framework for integrating a community of agents in a distributed environment. We will discuss the OAA in the section on related work (4.1).

Some critique has been voiced of agents as an interface tool. Shneiderman's view [66] is that agents are only as intelligent and useful as their creators make them and they will have to be very carefully designed to amount to any usefulness. He mentions several reasons why agents

may even be detrimental to the user/system interaction, one of which is that the user may loose control and reliability because of the delegation to the agent. The alternative given by Shneiderman is to use advanced user interfaces that are comprehensible, predictable, and controllable and that make use of new techniques for visualizing and manipulating the huge amounts of data that are becoming available to users. To the author of this thesis it seems that this argumentation is reasonable and that user interfaces *should* be predictable, controllable and directly manipulated whenever possible. However, there are problems with direct manipulation, such as dealing with

- information that is not available at the present time
- tasks that should be performed in the future
- a mass of information so large that the manipulation by a user is impossible
- sub-parts of some task that are numbered in the tens of thousands and therefore difficult to manage using direct manipulation
- filtering or structuring of information, when the user must be involved in the process

It is in these situations that agents will fill a need.

Given that we must use agents to perform some task or set of tasks, the question then becomes, how will the user-agent interaction take place?

As we will see in later sections, sicsDAIS provides a framework, such as the one described above, that enables interaction with multiple agents in one interface. In sicsDAIS it is possible for an agent to visualize itself to allow the user to contribute in the tasks of the agent, either using advanced anthropomorphic figures or simple configuration panels.

2.4 Demands of multi-agent systems in terms of user interaction

In the previous section, we examined the concepts of an agent and agent systems. As groundwork for the design of , we now state the demands of multi agent systems in terms of presenting to and interacting with users as the following:

1. *Access.* Agents that need to interact with users must be able to present themselves to users. Some agents provide services to other agents only and do not interact with users, and this type of agent only has to conform to the commonality in the communication procedure to cooperate within the agent system. Any information that such an agent provides which is of service to the user will be presented via other agents.
2. *Flexibility.* Agents provide different services and will want to provide different user interfaces. As is the case with legacy applications, the user interface of an agent is a reflection of the functionality of the agent and the service it provides. The methods it uses to communicate and interact with the user will depend on the modalities required, the type of the agent, the creator of the agent, etc. An alternative to this approach is to have a common interface for all agents. However, this places constraints on the agents. All agents would have to specify their user interaction in terms of the smallest common denominator of the common interface and it would be difficult to keep the system open to allow new agents to be introduced. This would also keep the agents from presenting their information in the best possible way since the common interface would constrain the interaction of agents that need specialized interfaces for their services.
3. *Dynamics.* Agents should not have to be constrained by having to conform to a hard-coded user interface. This is related to the previous, where one user interface is common for all

agents. If this is the case, the interface is static and services are hard-coded to the application. It is then difficult to foresee the dynamic needs of the system and to keep the system open.

4. *Multiplicity.* It should be possible to combine multiple agents in one interface. Sometimes it is of value to interact with several agents simultaneously (when agents need to share data; when one agent provides a service to another) and for this purpose, the dynamic combination of agents in one place is useful.
5. *Openness.* The agent system as well as the facilities for providing access to presentation and user interaction should be open (based on 2, 3, and 4). This will allow new services and agents to be introduced and removed during the course of running the system. It should be possible to add components of varying types in respect to the services provided, the implementation, and origin in respect to the service provider or source of the service.
6. *Adaptivity.* Adaptivity is desirable for tailoring the interaction experience to the individual user or group of users. The user interface facilities should provide means for adaptivity in two respects. Firstly, the interface must allow for changing characteristics that are based on the individual. This means that the interface should be able to present services and agents generically but also customized to suit the user. This could be in terms of sizes and colors of interface components, but also in terms of the modalities or methods chosen. Secondly, to be able to adapt to the user, the interface must provide feedback of the users actions to the underlying agents. The agents can use the feedback to build a profile or model of the user and determine the most appropriate mode of interaction.

This concludes the introductory sections of the thesis. We have covered a broad base of background information concerning presentation systems, agents, multi-agent systems, and interaction with agents. In the next section, we will begin examining sicsDAIS.

Chapter 3

The design of SICS Dynamic Agent Interaction System

Previously, we focused on the problems of agent presentation and agent-user interaction. Let us now describe sicsDAIS, the rationale of its design, and how this system addresses the problems discussed in the previous section.

We will start in section 3.1 with a statement about the philosophy of sicsDAIS. In section 3.2, we will examine a usage scenario in which an agent interacts with a user within the confines of an agent-based system. Section 3.3 describes Presentation/Interaction-descriptions (P/I-descriptions) used to construct interaction sequences in sicsDAIS. Then, in section 3.4, we describe the architecture of sicsDAIS. In the remaining parts of this chapter, we discuss the properties of sicsDAIS that make it a strong alternative to the previously mentioned systems. Finally, in section 3.9, we discuss how sicsDAIS attempts to provide for the demands noted above in section 2.4.

3.1 The philosophy of sicsDAIS

sicsDAIS is a unifying platform for interaction with agents. It provides the means for multiple agents to present information to the user and receive input from the user. It also provides means for agents to communicate and share resources and data.

Agents provide their own interfaces to sicsDAIS as smaller versions of themselves, much as mobile agents, and sicsDAIS coordinates the presentations of these.

sicsDAIS is an example of a model of one interface for many agents. It is the central point where the user interacts with all agents, but it is not a pre-defined interaction, since agents can dynamically come and go, and the methods of interaction can change.

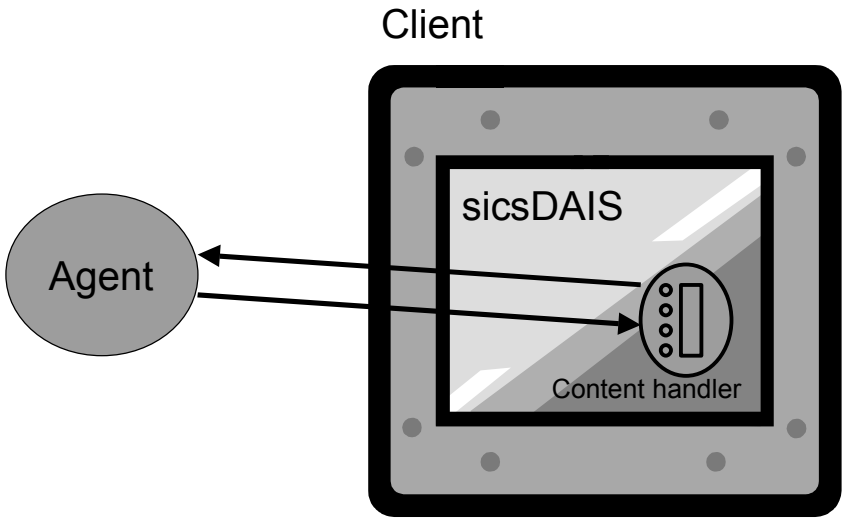

Figure 4. An example scenario of using sicsDAIS to interact with an agent. The agent is represented in sicsDAIS by the content handler.

3.2 Using sicsDAIS

We turn now to a scenario wherein sicsDAIS is used for facilitating the interaction between a user and an agent. The agent is part of some multi-agent system and sicsDAIS resides with the user and acts as the user's interface to the agent world. Figure 4 is a graphic representation of the setup.

The agent on the left in Figure 4 wishes to interact with the user. The user is the client on the right in the figure. The agent does so by establishing contact with the user's sicsDAIS and making it present the agent's content handler (or content handler*s*). A content handler is a piece of code made to visually represent the agent in the user's interface. It provides a presentation and interaction medium for the agent in relation to the user. We will examine content handlers more closely in section 3.5.

The content handler class file is dynamically loaded into sicsDAIS (much as the plug-ins described previously). Then the content handler is created with accompanying data from the agent that specifies visual and internal properties of the content handler. Finally, the content handler appears in the user's sicsDAIS as a separate component, much as the agent has intended[9]. The content handler of the agent knows best how to present the data originating from the agent since the content handler itself originated from the agent. At this point, the content handler may establish contact with the agent, thus becoming an extension of sorts to the agent.

After the content handler has been created and presented to the user, the user can interact with the agent through the content handler. Some of the interaction will be local to the content handler, i.e. not requiring any agent communication, if the necessary knowledge of the

[9] As we will see later, it is sometimes necessary to override the wishes of the agent in the interest of the total interaction with the user.

interaction and the required data is present in the content handler. In other cases, the content handler will convey queries to the agent. The content handler will exist in sicsDAIS for as long as the agent and/or user finds it necessary and may in the meanwhile make use of resources in the system or collaborate with other content handlers to accomplish its task. At the end of such a session, the content handler will be terminated and the agent has the option of starting a new interaction session by requesting the creation of a new content handler[10].

There may be several agents involved in a session with the user. They may all have content handlers active in sicsDAIS and they may each have several content handlers. Some sessions may be governed by a central coordinating agent that preprocesses presentation requests of other agents before they reach sicsDAIS and the user. In this way, the central agent can control the layout and other properties in sicsDAIS. In other cases, the presentations may be of a simpler nature and sicsDAIS itself may be capable of coordinating the requests of multiple agents. Whatever the scenario, sicsDAIS always acts as the interaction system, providing the necessary interaction services through the integration of agent-provided content handlers.

To illustrate the involvement of different parties in making an interaction sequence, let us examine the process of creating and implementing an interaction sequence using sicsDAIS and the different actors that are involved.

(1) There is the agent builder that programs the agent. (2) The content handler class creator (this may be the same person as (1)) programs the content handler class to be a multi-purpose content handler that may be tailored to suit many situations. (3) The interaction/presentation designer will consider the data and the interaction that the agents wish to present or achieve, and will choose appropriate content handlers for the task. P/I-descriptions, collections of data that describe the content handlers (see section 3.3) will be created that format the general content handlers to the task at hand (by specifying the parameters and events of the content handlers and thereby specializing them).

We will return to the different roles in section 3.8.2. We will take a closer look at the integration of content handlers in sicsDAIS as well as at the services sicsDAIS provides in the sections that follow.

3.3 Presentation/Interaction-descriptions in sicsDAIS

Complete presentations and interaction sequences are constructed using combinations of content handlers. The information to be displayed changes in each presentation but the same content handler classes can be used repeatedly. Each time a content handler is called on to display some piece of information, the information in question is incorporated into the content handler as the content handler object is being created. To describe which content handler classes to use for each piece of information, as well as the information itself, there is a need for *Presentation/Interaction-descriptions* (P/I-descriptions).

A P/I-description, created by a presentation/interaction designer, contains a description of the hierarchy of content handlers that can render the display. In the hierarchy, all non-leaf content handlers are of type *composite*, while the leaf content handlers may be derived from either of the content handler base classes.

[10] Garbage collection in Java will remove any unused objects (content handlers) when they are no longer active and referenced by any agents or content handlers.

The P/I-description also contains the properties that determine the way each content handler renders itself. Finally, if some specific behavior is required of the content handler, there are scripts mapped to events that may fire in the content handler. When a certain event occurs in (is detected by) the content handler, the corresponding script is executed.

A P/I-description is really a reusable and pre-constructed configuration of a view or display of content handlers. It specifies which content handlers to use, their relative layout, and their interdependencies. It may be augmented with specific run-time data, which additionally specifies the *current* configuration data, i.e. data for the current instantiation of the P/I-description.

3.4 The architecture of sicsDAIS

sicsDAIS is made up of a number of components (see Figure 5). The ComLayer (communications layer) for example, handles the communication with the agent world, and the layout engine handles the layout of the interaction elements (content handlers) in the presentations. The components are connected to the event handler (section 3.7), which handles the messaging between all other components and the ComLayer and the agent world. Each component can thus receive and send messages to and from agents and one another. The other components are

- *Content handlers*. We will examine the content handlers below.
- *The domain object database*. The domain object database stores data that may be shared by agents and content handlers. Any such component may register interest in specific data and will be notified when it changes or is deleted.
- *The exception handler*. Handles errors and exceptions that occur in sicsDAIS.
- *The property handler*. The property handler stores and makes available properties specific to the current instance of sicsDAIS on the client system.

Let us now go through and discuss each of these in turn, starting from the top with content handlers.

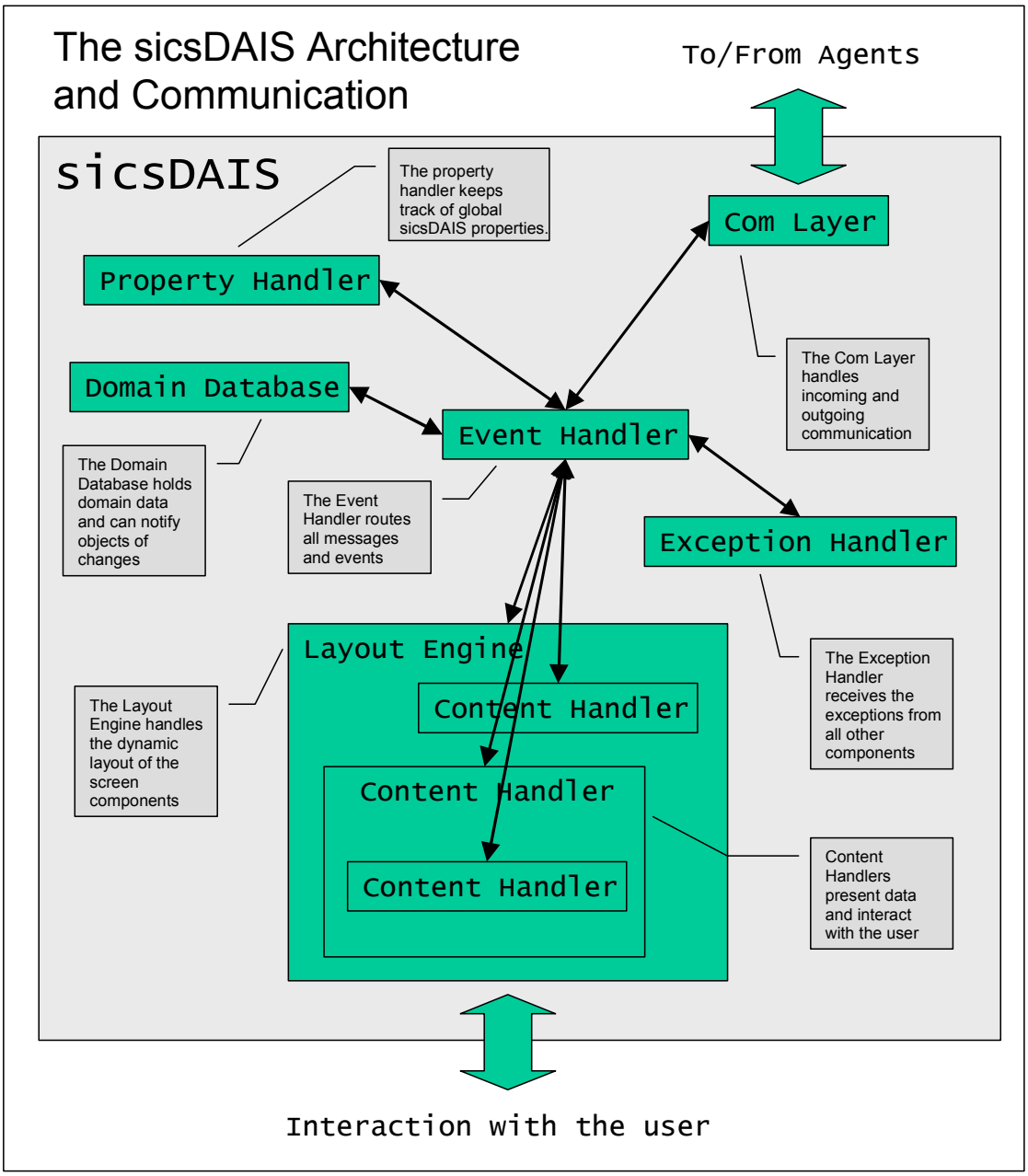

Figure 5. The architecture of sicsDAIS. The layout engine is the central component handling the presentation of the content handlers. The event handler dispatches incoming events (messages) to the correct recipient.

3.5 Content handlers

In sicsDAIS, the layout engine performs presentations using content handlers (we will discuss the layout engine further in section 3.6). A content handler is an object that represents an agent in the interface to the user. It is created from a class file that is dynamically loaded into sicsDAIS at the time of presentation. Each content handler handles one type of presentation, including the interactions with the user. A content handler can handle a simple interaction unit like a button, or a more complex one like a specially designed choice display. Using P/I-

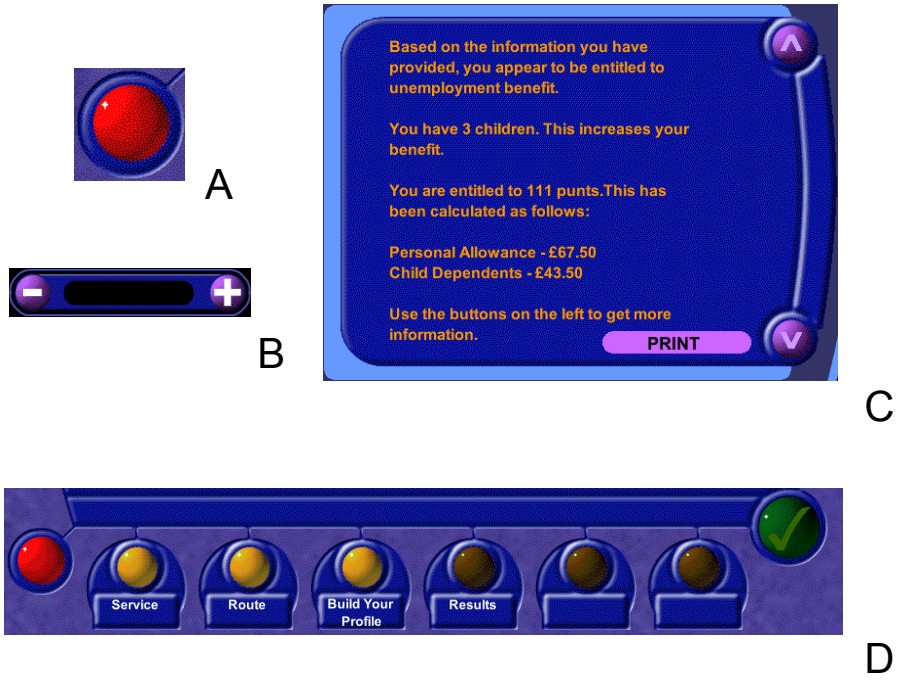

Figure 6. Examples of content handlers. Content handler A is an example of a simple atomic content handler, a button. Content handler B, a counter, is also atomic, although it is somewhat more complex in that it is possible to step through some enumeration by clicking the plus or the minus. Content handler C is another yet more complex atomic content handler. It allows the user to read and scroll through text. Finally, content handler D is a composite button bar made up of simple atomics (buttons).

descriptions that are sent to sicsDAIS it is possible to combine content handlers to create composite presentations[11].

As mentioned previously, P/I-descriptions contain scripts describing the content handler/s to use at a given point in a presentation or interaction sequence. A script is a description of a set of actions to perform within a given context. When creating an instance of a specific content handler, the script for this will contain the instruction to create a content handler as well as the name of the content handler class to use. It will also contain a number of commands that will be executed as method calls inside the content handler once it has been created (see section 3.8). These may set the state or attributes such as the color or the text that should be displayed in the content handler (see section 3.3). The scripts can in some cases be viewed as declarative descriptions as well as commands. In the following example, the attributes `layout` and `preferredLocation` of some content handler are described:

[11] A presentation that combines several content handler objects in a P/I-description is called a complex or *composite* presentation. It actually results in the implicit creation of a composite content handler for containing the other content handlers. See section 3.5.3 about the composite content handler.

```
'(layout "vertical")
'(preferredLocation 0 0)
```

Content handlers are Java classes that are provided by the content and service providers as a means for sicsDAIS to present the content in the best possible way. A content provider creates a content handler class by sub-classing one of two template content handlers classes provided with the sicsDAIS distribution: the atomic (AtomicCH) or the composite (CompositeCH) content handler.

The reason for this is twofold. The first reason is to make the class fit into the implemented architecture of sicsDAIS. The base classes contain necessary system methods that are used when administrating the content handler while it resides in sicsDAIS. These include methods for starting and stopping as well as for creating new content handler objects. The second reason is to gain access to methods for accessing resources inside sicsDAIS. The methods available in this category include

- Methods for communication with components inside and agents outside of sicsDAIS.
- Methods for access to shared data in the domain object database.
- Methods for specifying properties of the layout of the content handler, such as padding, alignment, and sequence. These methods are only available in composite content handlers (atomic content handlers cannot contain other content handlers and can consequently not perform any layout).
- Methods for getting information about the content handler itself (current size, for example) and the surroundings in sicsDAIS.

However, there are no methods or built-in functionality for creating the user interface of the content handler. The content handler providers must implement the interface and interaction methods that are needed and this is often the only functionality that they implement. The methods available to content handler authors are listed in Appendix B: Methods of content handlers.

In Figure 6, we see four content handlers labeled A through D.

3.5.1 Java

The Java programming language has been inspirational in the work of building the content handlers. We will in the next three sections look at Java applets, content handlers (in a web browser), and the AWT (Abstract Windowing Toolkit).

Applets

Java applets [6] seem to have much in common with the sicsDAIS content handlers. They are small applications written in Java that are downloaded from remote servers for execution in the client's machine. They execute in a virtual machine that may be incorporated into the user's web-browser or operating system. In the running environment (the virtual machine) applets have access to the windowing system of the hosting machine, and can thus present information as well as interactive elements to the user through a GUI, much as any ordinary application on the desktop. The important difference between the applets and regular applications is that applets are downloaded without concern for installation on the client machine, or updating the applets when new versions are made available (the downloaded applet is presumed to be the latest version). Of course, the downloading of foreign and often unknown code raises certain security issues such as protecting the user from malevolent applet

behavior in terms of access to files or private information. These concerns are dealt with in the virtual machine by using special security managers that prevent any unauthorized access.

There are also clear differences between applets and content handlers. Applets are not designed to be components in an agent architecture—content handlers are. Content handlers can communicate with agents using KQML. Applets are usually constructed individually without any provisions being made to allow for cooperation between them. In fact, applets are completely separated as they execute in the virtual machine, except for the public methods and attributes that they may provide. Therefore, if two applets are in the same virtual machine, and they are aware of the public access to each other, the two are able to communicate and cooperate.

Applets have no abilities given to them because of inheritance from the applet class (besides methods for performing as an applet). Content handler classes on the other hand inherit several methods for communication, scripting, etc (see "Appendix B: Methods of content handlers" for information on these).

Java content handlers

A web browser is built with capabilities for handling certain types of content like HTML, text, GIF images, etc. It is possible, however, to complement the built-in capabilities of a Java-based web browser by dynamically loading *content handlers*. A content handler is associated with a certain type of content (like plain text) and will handle that content in a content specific way.

When a web server publishes content of a new type, there are two ways of allowing web browsers to handle this new type. Either the browser is replaced by a new version, which can handle the new content, or a content handler for the new type is used. Along with the new type of content, a content handler for the content is provided by the server. As the browser loads the content, it discovers that it is of a type that is unknown (by looking at the MIME[12] [55] type of the content). The browser then tries to load a content handler from the same web server.

There is obviously a resemblance between Java content handlers and the sicsDAIS content handlers:

- They are built in Java. This ensures that they will function on all Java enabled platforms.
- They handle specific types of content.
- They are loaded dynamically. When the need arises the core functionality of the browser or sicsDAIS is complemented with that of the content handler.

There are also a number of notable differences:

- Content handlers in sicsDAIS are not tied to MIME types. A sicsDAIS type content handler can handle a specific type of content (e.g. a video stream), as well as a specific type of interaction (for example a specialized content handler for making database queries).

[12] Multipurpose Internet Mail Extensions. The type of some data may be described according to this standard. Examples are text/plain, text/html, and audio/x-mpeg.

- A sicsDAIS type content handler can communicate with other content handlers. This enables the creation of complex content handlers assembled from simpler ones.
- sicsDAIS type content handlers can make use of the domain object database for sharing of information.
- A sicsDAIS type content handler can be built to locally handle calculations related to the type of interaction it performs with the user.
- Finally and most importantly, a sicsDAIS type content handler may communicate with an agent outside of sicsDAIS.

Java AWT

The Abstract Windowing Toolkit (AWT) of Java [8] has been inspiring in many ways. The general idea of the AWT is to provide a platform independent API to the window toolkit. A Java programmer can write user interface code once and it will execute on all platforms that support Java. This is accomplished by mapping the Java layer of the API to the platform dependent API of the windowing toolkit. Each Java implementation of each platform provides a mapping to the platform specific widgets that are available. The result is that the interface will be constructed using the interface components that exist on the specified platform, a button for example will look like a typical Windows button in Windows and like a typical Macintosh button on a Mac. The mapping is done using *peers*, code components that implement calls from the Java language to the operating system specific windowing toolbox calls.

The AWT has inspired a number of design choices in sicsDAIS. The concept of content handlers in sicsDAIS corresponds both to content handlers as mentioned above, and to AWT components in Java. Each content handler in sicsDAIS is like a Java *Component* in that it is one component in the user interface. Content handlers can also be assembled and entered into composite content handlers, much as *Component*s in Java can be put in *Container*s. In Java, a Container is specialized as compared to the Component super-class in that it is able to hold other components and in that it can be assigned a layout manager to manage the layout of the enclosed components. In sicsDAIS, the composite content handler class provides the same sort of container functionality and the layout methods vertical, horizontal, and absolute (as described in section 3.5.3).

3.5.2 The atomic content handler class

The atomic content handler class is the base class of any simple content handler that an agent might use to present information. Atomic content handlers are typically simple buttons or menus, but can also be more complex constructions such as file dialogs or color pickers (a mechanism for choosing color by blending the three primary colors). They are created by agent providers by sub-classing the `AtomicCH`[13] class and adding any code necessary for the specific content handler (such as interface code).

---

[13] The name and the sub-class `AtomicCH` is used to distinguish end-user methods from methods in the super-class `Atomic` which are reserved for system use. This could be achieved using access modifiers in one class only, but the two classes are used for documentation purposes; the `AtomicCH` class documentation is available to content handler creators and the `Atomic` class documentation is not.

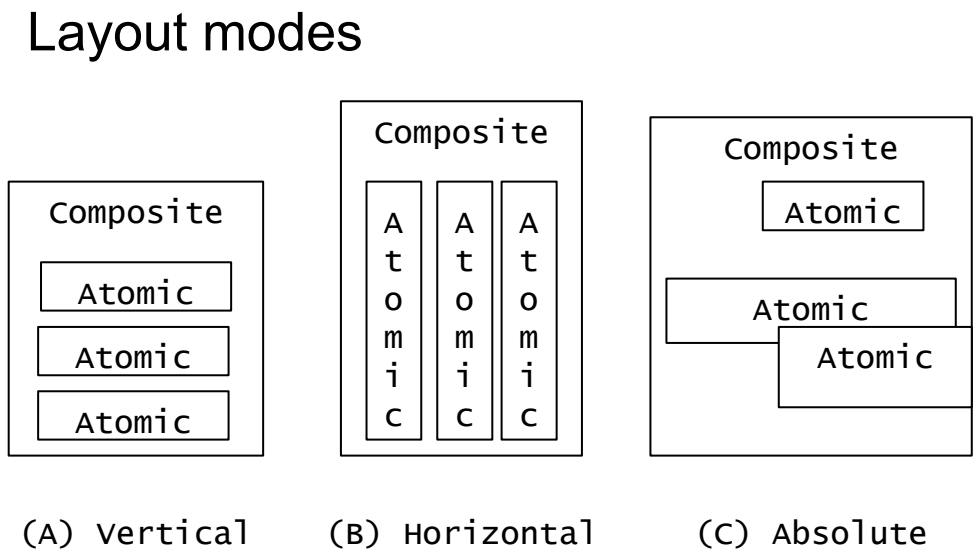

Figure 7. The layout modes of the composite content handler.

The following are characteristics of an atomic content handler:

- It really is atomic when instantiated in sicsDAIS; it is not constructed using other content handlers.
- It represents one unit in the user interface of sicsDAIS. As mentioned above, this may be a button or something more complex like a complete interface to an agent.
- It can receive messages from components inside or agents outside sicsDAIS. Messages can be sent to the atomic as a whole but not to any parts inside the atomic.

3.5.3 The composite content handler class

The composite content handler class (`CompositeCH`) is used in two ways. In the first, it is used similarly to the atomic as a template for creating new composite-derived content handlers, and in the second, it is used as a container for other content handlers.

A content provider that wishes to create a content handler, in which he or she uses other content handlers as components, can create a sub-class of the `CompositeCH` template and thus make use of its methods for layout etc[14]. This, however, is not the primary reason for this approach. If the layout functionality that is provided in the composite base class is insufficient for the purposes of the content provider, it may be necessary to refine, or even add to the methods provided. This functionality can be added transparently to such a sub-class by overriding or adding new layout methods. Thus, this is an exception to the rule that a content handler creator usually only adds the interface capabilities of a content handler.

The second use of the composite is for combining other content handlers, quickly and easily, to form composite presentations. This does not require sub-classing of the `CompositeCH` class;

[14] See Appendix B: Methods of content handlers.

instead, such a presentation is described using a P/I-description that specifies which content handlers that should be included. In sicsDAIS, the P/I-description is manifested as an instance of the base composite content handler class, with the specified content handlers as its components. Such a composite content handler has no appearance or interactive functionality, and it is not registered as a message receiver, but it provides the layout and container functionality for grouping the other content handlers. The layout functionality provided is rather limited when compared to a system such as LayLab (the experimental layout manager of WIP) [25]. This system considers the automatic layout a constraint satisfaction problem that is solved by finding an optimal solution.

The reason for the relative simplicity in sicsDAIS is that its focus is to provide a platform for interactivity and for presentation of information, while placing a minimum of demands on agents. In WIP, presentations are described using a declarative description of constraints, which is used to dynamically construct the presentation. sicsDAIS allows agents that so wish to construct presentations and interactive sequences to exact specifications regarding the layout. One could envision a content handler with the encoded ability of WIP to perform dynamic layout on behalf of agents, but in sicsDAIS, if presentations were required to be expressed this way, agents would be much too confined.

A composite acts as a container, and it is through scripting that a method for layout and other characteristics of the layout are chosen for the set of content handlers that it contains. The layout schemes that are available at present are *vertical*, *horizontal*, and *absolute*. These methods were chosen for their simplicity but as we have mentioned, it is possible to complement them with more advanced schemes if necessary.

Vertical layout (A in Figure 7) places the components of the composite vertically with an optional alignment in a number of different directions. Horizontal layout (B) places objects horizontally into the composite from left to right. Both of these modes also allow the contained content handlers to be expanded in any direction to fill the available space. Absolute layout (C in Figure 7) allows objects to be placed at any coordinate in the containing composite. The methods of layout can be combined by nesting composite content handlers.

Related to the laying out of content handlers according to some method is the use of adjustable sizes in content handlers. Each content handler class is required to provide methods stating the minimum and preferred sizes of the content handler. The composite content handler will attempt to distribute the contained components according to the layout method and the preferred sizes of the components. If this fails, the composite content handler may adjust the sizes of the enclosed content handlers to achieve the complete layout. The result is a layout that may be different from the agent's design but at least feasible when considering other content handlers that may already exist in the interface. This is the reason, as we mentioned in section 3.2 on using sicsDAIS, that layouts may not be rendered exactly as the service provider has intended.

The following are characteristics of a composite content handler:

- It may contain other content handlers combined together according to some layout scheme.
- Each content handler inside a composite content handler is itself a fully qualified content handler (unless of course it also is composite and P/I-description-created as described above).
- It may be an object of a sub-class or of the base composite content handler class itself.
- It may or may not be able to receive messages depending on the point above.

3.6 Components in sicsDAIS

We now turn to the remaining components of sicsDAIS:

- The *layout engine* that manages the layouts
- The *domain object database* for storing and sharing of data between content handlers as well as agents
- The *exception handler* for centrally handling errors and exceptions
- The *property handler* for client specific property handling
- The *communication layer* for communicating with agents
- The *event handler* that connects all other components in a messaging system (section 3.7)

3.6.1 The layout engine

The layout engine is the internal component that handles the layout of the content handlers that are placed on a root level in sicsDAIS (as opposed to content handlers that are placed within composites). The LayoutEngine class is a sub-class of the composite content handler and can therefore position elements in the interface of sicsDAIS according to the parameters available in the composite (see section 3.5.3 about the composite content handler). It can also receive messages (more on messages in section 3.7).

The layout engine handles two types of messages, messages that request new presentations of content handlers, and messages that modify its behavior. In the first case, it evaluates the incoming message and creates content handler objects according to the contents of the message. These are entered into the layout engine as components and laid out according to the parameters that are set for the layout engine itself. In the second case, the message changes these parameters. For example, the layout method used may be changed from vertical to horizontal, or the padding between components may be altered.

The layout engine is essentially no different from any composite content handler object, except that it is created as an internal component of sicsDAIS and as such is available to service agents from the start of a session.

3.6.2 The domain object database

The domain object database, like the layout engine, is an internal component in sicsDAIS. Its purpose is to store and make available any kind of data that may be useful during the course of a session of running the system. This includes data used by content handlers as well as data used by agents outside of sicsDAIS, but it is only used for domain data and not for data concerning the inner workings of sicsDAIS. All content handlers and agents are able to access the database since the database is scriptable.

For example, an agent can add data to the database that is later used by all content handlers of a certain type.

The reason for providing a domain object database in sicsDAIS is threefold:

- *Data sharing*. Both content handlers and agents can share data with each other. Furthermore, an agent can share data with another agent's content handler and vice versa. The sharing may take place while both components are actively performing a presentation

in sicsDAIS, or in a delayed manner where one component installs shared data that is later retrieved by another component.

- *Data storage*. A component (content handler or agent) may store data in the database for later use. In this way, the database can function as a cache for storing for example multimedia data, or as a profile memory for storing data about a user during the course of a session.
- *Communication between components*. An agent can change data that several content handlers are subscribed to (see the section on subscription and notification below) and thereby communicate the new data in an efficient manner. The updating of the data is performed by the agent sending a message containing the data to the domain object database. The database will then handle the notifications of the interested (subscribed) content handlers. The internal communication of sicsDAIS is more efficient than communication from agents to sicsDAIS, and a performance gain is achieved.

An advantage of this type of data sharing and communication is that it can occur between unrelated components as long as the identification of the data is known to the involved parties. They do not need to be aware of each other to share and communicate data. This increases the openness of sicsDAIS, as it makes it easier to introduce new agents into the system.

If the components are coordinating their efforts in presenting to the user, they will most likely have agreed on the names of the data that will be shared. However, one can also imagine that some data are stored under names that correspond to an ontology (an agreed upon structuring of concepts and their semantics) [26] or common language use. In this case, it is possible for the components to cooperate despite the fact that they are unaware of each other and the identification of the data. For example, an agent that needs the user's address could search the database for the object *user* with property *address*. If it is there, it may be used, if it is not, the agent will have to ask for it. This also helps to contribute to the openness of sicsDAIS.

Subscription and notification

Agents and content handlers can add, delete, and change information in the domain object database. For each piece of information there is a vector keeping track of which components that are interested in it. A component can register its interest in a piece of data by subscribing to it. As soon as the data is changed or deleted, all subscribers are notified through the usual event handling system. The domain object database will not delete any information as long as there still are subscriptions to it.

3.6.3 The exception handler

The exception handler is the part of the system that is responsible for keeping track of any errors or exceptions that occur. All parts of sicsDAIS are connected to the exception handler using the exception handling system in Java. This includes all content handlers as well, even those that have never been seen by sicsDAIS before. This is due to the fact that all content handler classes inherit the exception handling from the content handler super classes (the template classes they are based on).

If an exception should occur, it is caused either by a programming error in the code, or by a logical error in the transactions of the presentation. A programming error can be caused by faulty code in a content handler class or by faulty code in one of the system components. It is important to note that even code that is not within direct control of sicsDAIS (i.e. content handlers that are loaded during run-time) is controlled in terms of error handling.

A logical error can occur as the result of some misconception on the part of a content handler or an agent. The component might expect a certain piece of data to exist in the database when in fact the data has been removed. Another example is a content handler that tries to communicate with an agent that is off line. These types of errors are harder to anticipate and to deal with, and precautions have to be taken to prepare for such events. Since the exception handler is scriptable, it is possible for agents to set the appropriate behavior by sending it messages. It can for example be configured to notify a logging agent every time the event handler is unable to deliver a message because the recipient is missing.

3.6.4 The property handler

The property handler is used as a central point for storing session and client specific properties that may be of use to the other components of sicsDAIS. While the domain object database is used by agents and content handlers for sharing data, the property handler is used for handling properties of the sicsDAIS application. Properties such as values for the size of the main interface window, the default colors of content handlers, or sicsDAIS' ID on the network, are loaded from a file into the property handler at the start of a session. Each component and content handler can access the properties by using the `getPSProperty` method.

The properties are separate from the domain data for protection. In the domain object database, any component can enter or change any data. Properties on the other hand, must not be changed, and sicsDAIS must be assured of their integrity.

Properties are specified as key-value pairs in a property file that is loaded into sicsDAIS at system startup. Additional properties and changes to existing properties may be specified on the command line when starting the system.

3.6.5 The communication layer

The purpose of the communication layer or *ComLayer* is to handle the communication between sicsDAIS and any agents that use it. The ComLayer's interface towards the agent world comes in three different flavors, each specialized for the type of connection that is applicable. The first one is the socket interface, which is used when connecting the system via a socket to some other socket connected component. The second is the file interface that is used mainly for testing sicsDAIS, but also for using it as a viewer, to preview P/I-descriptions stored as files while constructing them. The third and final interface is the Agent Services Layer (ASL) interface, which connects sicsDAIS to the ASL. The ASL is a communication architecture for agent communication and management [45] used in the KIMSAC system. More on this in section 5.1.1.

The interface to use is specified as a property in the properties file or can be specified on the command line when starting sicsDAIS.

3.7 Messages and events

sicsDAIS' internal structure is built around a number of components connected through an event passing mechanism. The events are processed in the event handler and shipped from the sender to the receiver. Incoming messages from agents go through the event handler as does outgoing messages from content handlers and inter-sicsDAIS messages between the components. The event passing mechanism is completely threaded (each component has its own thread, thereby executing in parallel with the others).

Upon creation, all components in sicsDAIS are registered in a registry in the event handler. The registration is the process of entering the message inbox of the component in the registry table, along with the identity of the component. For sicsDAIS system components, the ID is the name of the component (for agents to communicate with specific components they need to know this name), and for content handlers it is the unique number assigned to them at the time of creation. The uniqueness of the names in the registry ensures that there can be only one recipient for each handled message. Each component in sicsDAIS is also assigned an outbox for putting messages in the queue to the event handler.

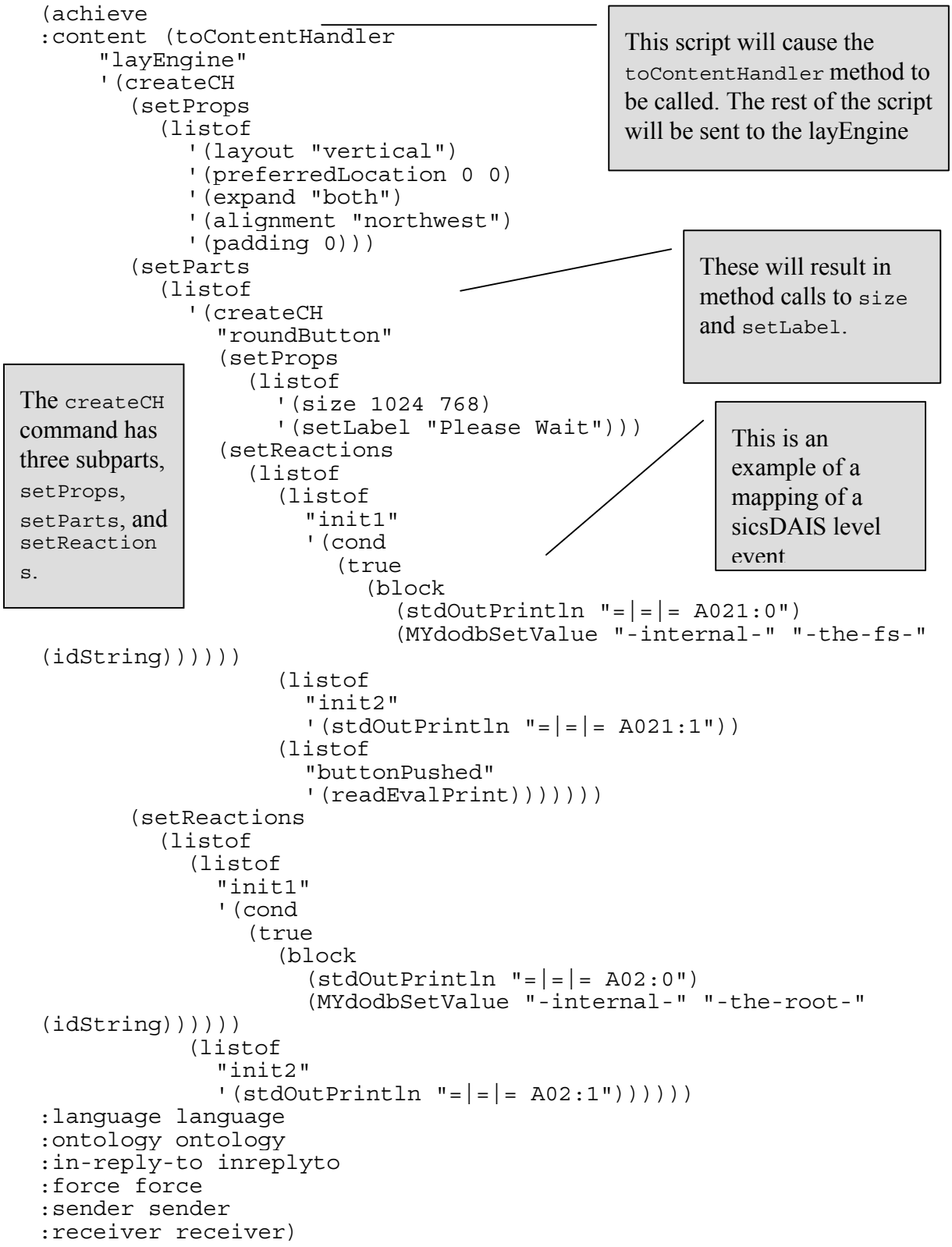

Figure 8. The message structure used in KIMSAC[15]

[15] Note to KIMSAC: the method previously named `createAsset` is now named `createCH`.

Content handlers and agents use KQML messages to communicate with one another. Messages sent by content handlers may be pre-stored in the content handlers prior to loading into sicsDAIS, or they may be created on the fly.

Incoming messages have a content part that is a script. The script can be viewed as a declaration of the required state or as a script to be executed within the context of the receiving object. A KQML message also contains fields that describe the sender and the receiver as well as the language of the content and other information[16], but it is only the content part that is eventually sent to the recipient. Conversely, messages that are destined for components outside of sicsDAIS (agents) are packaged into a KQML message and sent to the communication layer for dispatch on the ASL (or the socket connection).

Messages may be directed to sicsDAIS or to individual content handlers. Messages for sicsDAIS are general messages either requesting the creation and presentation of new content handlers, and are dispatched to the layout engine, or the modification of data in the domain object database. These messages have to conform to the content format that sicsDAIS uses (see the section on component scripting, below and the section on P/I-descriptions, above). A message sent directly to a specific content handler may be used to update the internal state of the content handler or to cause some action to be performed.

Figure 8 shows an example of the message structure used in the KIMSAC system.

The event handler sends messages arriving in its inbox to the appropriate receiver. The receiver's ID is looked up in the registry of components and the message is put into the associated inbox. To send a message, a component puts the message in the message queue of the event handler.

The message box

The message box is a programming construct used to implement the mailboxes described above. It serves two purposes. For one, it synchronizes threads that run independently, to allow one thread to wait for another to provide some data that is crucial to the execution of the first. Secondly, the message box allows messages to be queued, so that a component that is delivering a message needs not wait for a component to collect. In sicsDAIS, both forms of usage of the message box construct are present.

3.8 Component scripting

All modules as well as content handlers in sicsDAIS are scriptable, which means that they can receive and process scripts in the form used internally. This script format is a language reminiscent of Lisp and at the same time KIF [22]. See "Appendix A: Script syntax" for the syntax of the script language.

The processing of a script is done in two steps. First the script is parsed and a corresponding object of type `PSScriptObject`[17] is created. A `PSScriptObject` is a class that embodies the characteristics of the type in terms of the syntax and some of the semantics. Had the content language been something else, a different type of object would have been created. The parsing process ensures the syntactic correctness of the data and separates the script into logical parts.

---

[16] In the example above (Figure 8), these parameters are dummy values.

[17] `PSScriptObject` is a sub-class of `KIFObject`. `KIFObject` embodies the characteristics of KIF.

In the second step, the `PSScriptObject` is evaluated which leads either to some internal setting of temporary information in the form of runtime variables or to the invocation of methods in the current context.

The following is a piece of pseudo-code representing the mechanism for evaluation (interpretation) of the parsed script, where the `PSScriptObject` in principle is made up of a hierarchy of lists containing a method name and a list of arguments, (method arg1 arg2 ...):

```
evaluate PSScriptObject
{
  for all arguments in arglist do {
    evaluate argument
  }
  If function is internal then
    call internal function with arguments in arglist
  else
    call dynamic method with arguments in arglist
  return result of call
}
```

In the context of the event handler, a message evaluation may result in the message being routed to the appropriate receiver. The receiver will be either an internal component or a content handler.

Messages that are sent to internal components are again evaluated in the context of the receiving object. The purpose of the message may be to update a value in the domain object database, if this is the recipient, or it may be to change the behavior of the exception handler in response to a certain exception. Since the internal components in the system are all scriptable, sicsDAIS is totally configurable at run time.

Here is an excerpt of the message in Figure 8:

```
(achieve
:content (toContentHandler
    "layEngine"
    '(createCH
      (setProps
        (listof
          '(layout "vertical")
          '(preferredLocation 0 0)
          '(expand "both")
          '(alignment "northwest")
          '(padding 0)))
        ...
```

In the example above, the symbol "`toContentHandler`" will make the evaluator call the method named `toContentHandler` in the current context. In this case, the context is the event handler. The parts of the script after the method name will be parameters in the call to the method. The parameters in the example are "layEngine" and a script starting with

```
'(createCH (setProps (listof '(layout "vertical")…
```

When the `toContentHandler` method is called, the first argument specifies a content handler that will be sent the second argument as a script. The quote in front of the script signifies that this part of the total script should not be evaluated in the current context. When the script is sent to the layout engine, it will be stripped of the quote and evaluated within the layout engine (which then will be the current context). Here it will result in the creation of a new content handler with the call to `createCH`. At the end of the example, there is a list of method

calls for various properties of the content handler: the layout is set to vertical, the alignment is set to north, and so on. These properties also result in calls to methods.

One last note about the example, the `createCH` method call is not followed by the name of the content handler that should be created. This signifies that it should be a composite content handler. This is as opposed to the example in Figure 8, which contains creations of other types of content handlers.

3.8.1 Dynamic calls in the evaluation of messages

sicsDAIS receives messages from agents telling it what to present and how to present it. Content handler classes are loaded dynamically into the system, as they are needed. That is to say, when an agent requires the use of a content handler, that content handler class is loaded into sicsDAIS by the Java runtime system. The information in the messages is in a form that the specified content handlers understand. For example, a message might specify that the content handler "button-row" should be used in a certain circumstance. sicsDAIS then loads the content handler class (assume it is called "ButtonRow1.class"), creates the actual content handler object, and then goes about configuring it according to the specifications in the message. This will involve calling methods in the content handler that was just loaded and created.

How can sicsDAIS call methods and thereby set parameters in a content handler that it has never seen before and that certainly was not available for inspection at the time of the compilation of the system?

The solution is to use Java's dynamic method calls [41] in the context of the content handler. This means that a method that is identified only through its name and the arguments it takes can be called in a never before seen piece of code. sicsDAIS never knows about the internal functionality of a content handler but it can still configure it the way the presentation/interaction designer or the author of the content handler class intends. The point of using dynamic method calls is to allow dynamic usage of the content handlers without placing any constraints on the content handlers in terms of what methods it must support. The result of this scheme is that content handler authors are given free hands to implement their code in any way they wish using whatever methods that seem most appropriate.

The use of the dynamic method calls clearly gives a great advantage in an open system like sicsDAIS. In fact, it would be difficult, if not impossible, to achieve the same kind of openness and dynamics without them. Without an open system with dynamic loading of content handler classes all functionality would have to be coded into the system from the start, thus making it impossible to add new agents with their own mode of presentation[18].

To sum up: agents send KQML messages to sicsDAIS to achieve presentations of content handlers. Each message contains a script that is evaluated in a component in the system. As a script is evaluated, some commands in the script result in dynamic method calls to methods in the component. When a component sends such a script internally, it is called an event. All the base components as well as the content handlers are scriptable which means that they are able to evaluate scripts and thus handle events. Components in sicsDAIS can also communicate with agents using the same type of KQML messages. In this case the content language may be any language that the agent understands.

---

[18] However, issues concerning performance have to be dealt with. We discuss this further in section 3.10.

3.8.2 Events and scripts

In Java, an event signals the occurrence of some action, either in the interface or somewhere else within the scope of the program. We will call this a Java level event. A program can react to such an event by listening to the component that is the originator of the event [8]. This involves writing code that registers as an event listener at the event-producing component. At the time of the event, the listener will be notified by a call to a specific method. This way of handling events is fine if you are the programmer and the creator of the content handler class. You can then specify exactly what should happen when some event occurs. The problem is that the actual user (as in service provider in need of presenting information using content handlers) of a content handler may not be the author. In that case, should the user of the content handler have to change the Java code to make the content handler behave according to his or her wishes?

In sicsDAIS we have created an abstraction to the Java level events, called sicsDAIS level events. These events may have any name and may result in any action. All the parameters of these events are scriptable and they are first set as the content handler is created. The purpose of the sicsDAIS level events is to isolate the mechanics of the Java event model from service providers that use content handlers to present information and services. This means that the *user* of a content handler can script sicsDAIS level events (tie events to behavior) that the content handler is capable of firing, without having to bother with the actual code of the Java level events.

In section 3.2, "Using sicsDAIS", we spoke of three participants in the process of creating an *interaction sequence* (essentially a domain application achieved using agents, content handlers and sicsDAIS as the interaction system). To recapitulate, these are (1) the agent builder, (2) the content handler creator, and (3) the interaction/application (P/I-description) designer.

What is interesting is that it is likely that the content handler class creator (2), and the P/I-description creator (3), are not the same. The content handler creator may create content handlers as one would create classes in a class library, while the P/I-description creators in this analogy would be represented by the application builders that use the class libraries. In fact, it is likely that the P/I-description creator uses content handler classes that are created by a completely different company or institution for a completely different task. Naturally, the main reason for this flexible division of the design labor is to allow for added reuse of content handlers.

The content handler class creator knows what task the content handler class should be able to accomplish. He or she knows what buttons or menus should be included in the interface and what events such elements can fire in Java. The author of the content handler class can therefore map the Java level events to arbitrarily named sicsDAIS level events that will fire as a result of the Java level events.

This mapping is done using the method call "`trigEvent(<event-name>)`", which when called causes the script associated to "<event-name>" to be executed, if such a script is bound in the content handler. The mapping of Java level events to sicsDAIS level events should be described in the content handler documentation produced by the author.

Let us consider an example. If a content handler is sensitive to "`mouseDown`" Java events, the content handler class author should make sure that the Java method call "`trigEvent("buttonPushed")`" is executed when a "`mouseDown`" event occurs. This is a standard Java method invocation. When the content handler is created as an object in

sicsDAIS, the sicsDAIS level event, "`buttonPushed`", may be bound to a script, for example the following script:

```
(sendKQMLMess "hol@ad3" '(start-session) 'msgId45)
```

This particular script will send a KQML message constructed from the parameters following the agent ID to the agent named hol@ad3.

When the content handler traps the `mouseDown` event, the `buttonPushed` sicsDAIS event will be fired (by an internal sicsDAIS mechanism) and the associated script (the one above) will be executed. This will cause a KQML message to be sent to the agent addressable as "`hol@ad3`", with content `(start-session)` and message id `msgId45`. The presentation designer only has to know about the name of the sicsDAIS event and which Java events in the content handler that cause it.

3.9 Evaluation of sicsDAIS in respect to the demands on interaction and presentation

In the introduction and background sections of the thesis, and in particular in section 2.4 about demands on agent systems, we spoke of desirable characteristics of such systems in terms of presentation and interaction. We will, in this final part of the design section, restate these properties and examine sicsDAIS accordingly.

3.9.1 Access

Agents must have access to presentation to, and interaction with, the user. sicsDAIS is a platform for providing this access. It allows agents that so choose to present information and interact with users.

3.9.2 Flexibility

The user interface facilities should be flexible, in terms of allowing agents individual modes of interaction, to enable the most efficient interaction possible between agent and user. sicsDAIS provides this capability using content handlers and by providing coordination and cooperation facilities for content handlers. Content handlers can be provided individually by agents to cater for specific needs of the agent, but they can also be shared between agents in construction of more complex interaction and presentation schemas.

Specifically, sicsDAIS can be used as the interface for a single agent or as the coordinating interface for multiple agents. In the case of the single agent, the system provides the communication and the database access for the content handler, and in the case of multiple agents in an agent-based application, the layout engine provides three layout modes. sicsDAIS can also house invisible content handlers that can serve the other components. It is even possible to create a server function inside sicsDAIS, which content handlers in other sicsDAIS instances can access.

3.9.3 Dynamics

The user interface should allow for dynamic changes in and the introduction of new methods and modes of interaction and presentation. The content handler architecture provides this. Agents can provide and integrate content handlers for any type of content.

3.9.4 Multiplicity

The user interface should be able to present multiple types and multiple concurrent instances of agents. In sicsDAIS, this is possible by combining content handlers from several agents. sicsDAIS provides functionality for

- Coordination in that content handlers can be organized visually on the screen according to layout schemes
- Cooperation in that content handlers can cooperate and react to each other using the event handling mechanism
- Sharing of data between the multiple content handlers using the domain object database

3.9.5 Openness

The presentation and interaction system of an agent-based system should be open. New services and agents must be easily introduced during the course of running the system, without requiring modification of existing parts. It should be possible to add disparate components created by varying vendors on different platforms using different development means.

sicsDAIS is a completely open system. Any agent can make use of it to present information to the user using content handlers. sicsDAIS has a communication interface that allows agents to send it requests for presenting content handlers, and for communicating with the content handlers in sicsDAIS once they have been created. As of yet, sicsDAIS does not address any security concerns beyond those intrinsic to Java (verification during linking whereby loaded classes are checked to conform to the Java Language Specification [24] and no direct access to physical memory, etc. [20]). There is no need because KIMSAC is a closed agent community. In future applications, the nature of the services provided may require unknown agents to be permitted to enter the sicsDAIS domain. In this case, these concerns will have to be dealt with using methods such as encryption and digital signatures[19].

Agents can come and go dynamically and the requirements on the content handlers that agents use to interact with users are minimal. sicsDAIS only knows about the content handlers that are instantiated inside it. If an agent goes off line, another agent may carry on the communication with the first agent's content handler. The only demand that is placed on an agent is that its content handler class(es) be derived from one of the base content handler classes, AtomicCH or CompositeCH, that sicsDAIS provides. This is a simple matter of sub-classing. The base classes only require that the interface code is added, methods for dealing with scripts and messages as well as for accessing the domain object database and the property handler are provided.

3.9.6 Adaptivity

In an intelligent user interface (IUI), in which a combination of different modalities is used to create an advanced interaction environment [52], the concept of adaptivity is very important. It is one of the factors that distinguishes an IUI from an ordinary direct-manipulated interface (among other things [37]). The fact that the interface, and through the interface the underlying system, can modify its behavior to the individual needs of a user makes for a more dynamic and constructive interaction.

---

[19] A digital signature verifies the authenticity and origin of some data.

What makes a system adaptive? A system is adaptive if it changes in some way in response to the user's actions for the reason of improving the interaction between the user and the system. The purpose of the adaptivity may be to adjust the interface to better suit the user, or it may be to tailor presented information to the user [35]. The reason for changing the appearance of the interface or the ways of interacting with the system is to make the interaction experience more efficient or pleasant to the user. This is done by taking into account individual properties of the user, like cognitive ability or poor eyesight and then changing the ways the user can interact with the system accordingly. The result may be a system that is easier and more efficient to use [3].

To change the information that is presented is to tailor the information to the individual needs of the user. This could mean filtering information according to the users interests or blocking out irrelevant information in certain contexts. Think of a student searching for material for writing a term paper. As the student uses the library search system, the system picks up queues about the task at hand and starts to tailor the search results to the student. The system could, after determining that the student is searching for academic papers, start to filter out popular science articles. Then, after determining the subject of the students interests, the system could present papers in related fields having some relevance to the main subject.

What are the requirements to *enable* a system to be adaptive? Mainly there are two. One is that the system is able to adjust itself dynamically in terms of the way it interacts during the course of the interaction with the user. It must be possible for example, for the system to adjust the font size if the system is going to be able to adapt to a user's eyesight. This type of requirement is mostly technical and is usually relatively easily built into a system. The second requirement is that the system knows the preferences or properties of the user. To be able to adjust to the user the system needs to know about the user. This may be done by allowing the user to specify preferences or by the system learning about the user during the interaction. This perception and understanding of the feedback of the user to the system is of course more difficult to achieve.

The purpose of this section of the thesis is not to elaborate on the theories of user modeling and adaptivity but rather to examine in what ways sicsDAIS is equipped to alleviate adaptivity. As stated above, there are two main requirements of a user interface to enable it to adapt to a user, the capability of the interface to adjust dynamically, and the capability of the interface to provide the feedback from the user to determine what the user is doing. sicsDAIS fulfills both of these requirements.

In sicsDAIS all presentations are created by combining content handlers that have been laced with user or context specific data. This dynamic generation of presentations is inherent to sicsDAIS but also the answer to the first requirement above, to be capable of modifications of the interface to suit the individual user. This is clearly feasible by changing the way the presentation is created by the agents responsible.

The second requirement is to allow feedback from the user to reach the underlying system. In sicsDAIS, as we saw in sections 3.5 about the content handlers, and section 3.7 about the communications architecture, all communication between the user and the agents goes through content handlers. All communication between components in sicsDAIS passes through the event handler component. This means that anything that takes place in the system, whether caused by the user or the system itself, may be logged. This information may also be transmitted back to interested agents in the background. Note the use of the word *may* here. It is completely up to the content handlers in each presentation as well as the responsible agents to make this happen. sicsDAIS only provides the means for doing it.

One example of using feedback to adapt the interface is the adaptive help agent that is described in section 5.1.3. Briefly, this agent is located inside sicsDAIS where it is sensitive to communication, as well as to changes of the data in the domain object database. In response to the user's actions and the current context, it can adapt help information to suit the user.

3.10 Performance

The subject of performance is interesting because of the impact it has on the presentation and interaction dynamics of a session of using sicsDAIS. The overall most important performance issue of sicsDAIS is the choice of the implementation language Java. Java is still an immature platform for software development in terms of performance. Optimization of interface rendering but also of processor intensive calculations is needed. In a system such as sicsDAIS, the major performance demands are those of fast interface rendering, prompt parsing and evaluation of scripts, and related to the latter, efficient dynamic method invocations. Some performance improvement may be gained from using a just-in-time (JIT)[20] compiler, but this generally has the most impact on heavy computation, which is the smallest problem in sicsDAIS. It will however effect the parsing and evaluation, but as we will see below, these are efficient as they stand. The interface rendering is technically difficult to make more efficient, without redesigning and re-implementing, the Java AWT with a specific and streamlined version for a specific platform. This is clearly unfeasible because of the work involved (as well as the fact that it counteracts the aim of sicsDAIS to be platform independent). We will in this discussion therefore concentrate on the dynamic method calls, the parsing and evaluation, and other issues such as the performance of the third party content handlers including the Task Manager (TM) and the Adaptive Agent (AA). We will then discuss another improvement in the design of sicsDAIS concerning caching of media. We will start with the matter of the dynamic method calls.

3.10.1 Dynamic method calls

We described the usefulness and importance of dynamic method calls in section 3.8.

However, the usage of the dynamic method calls has its drawbacks. The process in which a dynamic call is made starts when the method in question is searched for in the context in which it will execute (a sicsDAIS component or a content handler). The method name is matched against all methods in the object that bear the right name in an attempt to find one that not only has the right name but also has the right argument profile. This look-up may or may not succeed of course. In any case, the process of finding the right method takes some time so performance is at an issue.

The second disadvantage of using dynamic method calls is that the process of calling the method is very costly. The reason for this is the type checking that is done at the time of invocation. The actual parameters of the method call have to be checked and formatted (by wrapping in or unwrapping from objects) during run-time. The return value of the call, if a primitive type, must also be wrapped in an object.

There is no simple solution to this problem. Content handlers inherently have differing APIs and there is consequently no way to foresee all of the method invocations that are possible. One could possibly include statically linked calls to the most common method calls and use naming conventions for common methods like color, size, reaction, etc. This way only

[20] A JIT compiler translates the current instruction to native code before execution.

unusual method calls in never before seen content handlers would be affected by the performance penalty. This is in fact what is done with sicsDAIS in the KIMSAC system. Here, an analysis of the content handler classes yielded a set of methods that were then hard-coded into the `PSScriptObject` class.

3.10.2 Parsing and evaluation

The parsing is efficient in terms of speed and memory usage due to the use of the JavaCC [40] parser generator in the implementation. The evaluation is also inexpensive since there is little to evaluate after the parsing and the dynamic calls.

3.10.3 Content handlers

The inefficiencies of content handlers are twofold. First, third-party content handlers (content handlers not provided with sicsDAIS) may not be optimally implemented. This is a fact that is hard to deal with since there is no control over a decentralized development community where code is used indiscriminately at run-time. Secondly, the nature of content handlers is to provide interaction capabilities between agents and users through an interface. This interface may be more or less graphically or visually advanced with greater download times for media as a result. In an advanced GUI of a content handler, there may be video clips in the order of millions of bytes to load, and this takes time. One could possibly use a cache, but the amount and total size of the media used in a typical application and in KIMSAC in particular, may well reach and surpass 500Mb. Some caching of simpler and smaller media is done by the content handlers themselves but maybe a centralized handler for media loading and caching in sicsDAIS could improve matters. It is clear, however, that a dynamic system like sicsDAIS, for interaction with graphically oriented components, is bound to require a great deal of data in the shape of media. The only direct solution is to increase the bandwidth allocated to the network.

The Task Manager

The invisible Task Manager (TM) (see section 5.1.3) inside sicsDAIS in the KIMSAC system, is a content handler, loaded at the start of a session, responsible for the coordination of some activities in sicsDAIS concerning service agents and their content handlers. It is the source of some performance degradation. It is at times very processor intensive because of the amount of evaluation and parsing it must do to handle the complex descriptions of the interdependencies of the service agents and their content handlers. The TM is completely programmable through scripts and contains little actual Java code. Thus the non-proportional amount of computation needed for parsing and evaluating all the scripted code.

An alternative solution, given the problem that the TM is needed for some coordination that cannot be distributed on the service agents, is to provide a template coordination handler within sicsDAIS. This would provide some of the required functionality through a Java API, but it would be extendable to allow added functions according to specific demands of specific applications. It would provide services such as handling of interaction infrastructure (a menu of agents, for example, that is not tied to any specific service agent), start and shutdown of sicsDAIS sessions, application specific handling of exceptional and critical events (in cooperation with the Exception Handler).

Chapter 4

Related work

In this section, we will examine two systems for realizing agent applications, the Open Agent Architecture (OAA) and the Aglet Workbench. These systems are used to create applications that use agents as components. The main difference between the two is that the OAA is a facilitator or blackboard based system (all communication between agents goes through a central component named the facilitator), and the Aglet Workbench is based on mobile agents. These systems are interesting because they are based on agents and they aim to provide users with the type of delegated functionality that is not found in direct manipulated systems, and also because of similarities to sicsDAIS; the OAA has a design for its user interface capabilities that is similar to sicsDAIS, while the design of the mobile agents in the Aglet framework is similar to the content handlers in sicsDAIS.

4.1 The OAA

The OAA, created and constructed by Adam Cheyer, David Martin, Douglas Moran, and others at the Stanford Research Institute (SRI), is a framework for integrating a community of software agents in a distributed environment. The focus of this effort is twofold, to explore ways of building flexible and adaptable distributed systems, and to examine ways of user-interaction with the agent-based applications [57]. The central idea of the OAA is to promote the cooperation of the involved agents with adaptable and flexible interactions among the components through the delegation of tasks.

The OAA is open since agents can be written in many different languages and can be interfaced with existing systems. New services can easily be added. It is also

- *Extensible*. Agents may be added or removed dynamically.
- *Distributed*. Agent can be spread across many hosts.

- *Parallel*. Subtasks in OAA run in parallel if appropriate.
- *Mobile*. The OAA uses lightweight interfaces on mobile devices.
- *Multimodal*. The OAA uses handwriting, speech, gestures, and direct manipulation in combination.

The facilitator

The OAA is a framework as well as a toolbox for building agent applications. It is based on the notion of a facilitator, a central component that conveys communication between agents as well as plans and executes goals, which are described in the Interagent Communication Language (ICL) (see below). All agents are created or provided with a set of functions for accessing the facilitator, and they publish their capabilities with the facilitator. Part of this publication is the natural language vocabulary that the agents understand and that can be used to communicate tasks to the agents. Several systems of facilitator structures may be combined to form a larger construct. In this case, the facilitators will query each other when searching for ways of solving goals.

The Interagent Communication Language (ICL)

This is a logic based declarative language capable of representing natural language expressions. It is based on speech acts like `oaa_Solve` (a data request or procedure request) and `oaa_AddData` (data management). Horn clauses describe the content of the speech acts and are augmented with temporal constraints and time stamps. These are used to describe future events and to reason about whether the constraints are satisfied. The language of the horn clauses is common between agents. Agents must use terms that are close to natural language and there must be an agreement between them as to the meaning of the terms. For example, if an agent wishes to know the location of the user it may query another agent with the statement, `oaa_solve(location(user, U))`. In this case, both parties must agree over the term location and its arguments. The language needs not be fixed in advance; it needs only be common.

ICL is used to execute actions, perform queries, manipulate data, and exchange information.

Client agents

There are four types of client agents:

- User interface agents that interact with users
- Support agents that provide very general services used by other agents, for example natural language agents that understand natural language and translate this into ICL
- Application agents that provide services that may be either domain dependent or domain independent. Most application agents are created by wrapping a layer around a legacy application to make it OAA compatible
- Meta-agents that help to coordinate the efforts of the other agents by applying domain knowledge to the tasks at hand

All agents in the distributed environment contribute services to the community. When services are required by an agent, a request is posted with a facilitator that coordinates the efforts to solve the task. This is done by the facilitator determining what agents are able to solve what

tasks and then distributing sub-tasks to be solved. The results are compiled by the facilitator and presented to the requesting agent.

User interaction

Human users are a central part in this scenario. They interact with the agent community through natural language, which is translated by special language agents into high level queries in the ICL. Additionally, the OAA has incorporated a number of facilities to enable users to point, draw, write by hand, or use a GUI in the interaction with the agents.

The user-agent interaction is managed by a User Interface (UI) Agent. It manages the different modes of interaction and distributes the different inputs to other agents as needed. For example, when speech input is detected, a command is sent to the speech recognition agent to process the input. The result is returned as text that may then be processed further if necessary by a natural language understanding agent.

There are two ways of interacting. In the first, a GUI is presented for the application agent. This interface has been designed specifically for the agent and is presented in a window with familiar GUI-style items. The UI agent handles all other types of input (speech, handwriting, etc) which isolates the agent from the details of the modalities being used. This simplifies the introduction of new modalities and the design of the agents. This is quite different from the approach taken in sicsDAIS, and it has advantages. In sicsDAIS, each agent through its content handler(s), is responsible for interpreting input from the user. There is no common handling of different modalities. It would clearly be useful to have an on-board modality coordinator and a number of content handlers for handling different types of input. The strength of the system, on the other hand, is that it allows agents to make use of *any* means for the interaction with the user as long as content handlers are provided by the agents. sicsDAIS itself places no constraints on the interaction because of lacking tools for interpreting different modalities since it allows agents to dynamically submit code that handles their interaction.

The other mode of interacting in OAA is through the UI only. In this case, there is no application agent visible and the user interacts using the multiple modalities of the UI. Various invisible agents may be involved by the UI in interpreting the input.

The two modes of interaction may be combined, as we will see in the section below.

An example

The OAA has been used to implement a number of different applications. One is the automated office application. In this system, 14 agents provide communication and information handling services to users in an office environment. The system makes use of a multimodal user interface for presenting graphics, text, and speech, and for input through handwriting. Speech input is provided using a telephone interface.

In the system, the user is presented with a graphical “office” where clicking on a symbol portraying for example a wall clock will bring forward the interface of that particular agent. The GUI in this case is a static screen with hotspots that activate the underlying agents. The user interacts with the specific applications directly. In addition, there is a smaller window for speaking or writing natural language commands that will be handled by multiple agents. For example, the user poses a question through the voice interface, “where is my next meeting”, and a multitude of agents will be involved in producing an answer. The speech agent will translate the query into an ICL statement that will be posted to the facilitator. The facilitator

will involve the calendar agent as well as perhaps an agent that keeps track of locations in the office building, and will combine the resulting answers in making the response.

4.2 The Aglet Workbench

While the OAA focuses on the user interaction and the cooperative aspects of an agent community in terms of solving tasks and sharing information, the Aglet workbench [47]. is an example of an architecture for building mobile agent applications. It was developed at IBM's Tokyo Research Laboratory under the direction of Danny B. Lange.

An Aglet is a Java-based mobile agent that may move from host to host in attempting to gather whatever information necessary to complete its task. Aglets may move arbitrarily between hosts, as opposed to content handlers in sicsDAIS which move only from the providing agent to the client's sicsDAIS.

Aglets are built with the components mentioned in section 2.3.3 about mobile agents: a state, an implementation, interfaces, an identifier, and principals, but they also contain the following:

- Listeners. These are modules of code that an Aglet author can add to an Aglet to specify its behavior in certain situations. There are clone listeners for reacting before and after the duplication of an Aglet, mobility listeners for reacting to the movement of an Aglet, and finally persistence listeners that react before or after activation and deactivation of an Aglet.
- Proxy. This is a wrapper around the Aglet that regulates the access to the Aglet's public methods and provides location transparency. This means that the proxy can be in one location while the Aglet is in another and this is transparent to entities that interact with the Aglet.

The environment

The "place" is where the mobile Aglet is resurrected after having been moved from one host to another. The place is actually a server running on the host, which allows foreign Aglets to visit and execute. A place is made up of the following parts:

- An *engine*. This is a virtual machine that runs on the host machine and that allows Aglets to visit and execute.
- *Resources*. The place has resources that are made available to the visiting Aglets.
- A *location*. The place has a location (an address) on the network.
- *Principals*. These are the same as for the Aglet and they govern the usage and responsibilities of the place.

Aglets move from place to place in the following manner. For some reason, either internal to the Aglet, or external, the Aglet decides to migrate. It is made aware of this by a call to a certain internal procedure. The Aglet is then suspended in the current place, after having been allowed to save its state and perform whatever housekeeping procedures are necessary. It is then serialized (the whole Aglet including the state is converted into a byte stream) and transferred to the new host. At the new host, the Aglet is reinstated, by deserializing and restarting it.

Building an Aglet

To build an Aglet, a sub-class is made of an abstract Aglet class (abstract means that the class has hooks for procedures that the user of the class must provide) that is provided in the Aglet workbench. This class provides most of the functionality of the Aglet in terms of its existence within a place as well as for communication and migration. The implementer of a new type of Aglet must provide code that describes the behavior of the Aglet in terms of its specific purpose. That is, the code that provides the Aglet with its actual domain behavior. An example of this would be the behavior of an Aglet that moves around a network calculating and gathering information on bottlenecks in the network. The code for this measurement would have to be built into the Aglet by its creator.

Aglets are very similar to Java applets in that they contain the code necessary for execution and that they execute in virtual machines on remote hosts. As with applets, Aglets must provide their own interface to the user and there are no predefined interaction schemes. They are also similar to content handlers in sicsDAIS in this respect. Users will interact with Aglets that are created by multiple vendors, which means that no common interaction system will be available. This is an example of the interaction model of one interface for each agent (see section 3.1 on the philosophy of sicsDAIS), as opposed to the approach taken in sicsDAIS of one interface for all agents.

Aglets vs. content handlers

Aglets move between hosts executing in “places”, or virtual machines that exist on the hosts. This is similar to a sicsDAIS instance that may be viewed as a “place” in which content handlers can execute. The difference is that content handler classes are downloaded to sicsDAIS as a result of an agent wishing to interact with a user and it is only the interaction intelligence that is downloaded. This is a bit like the code-on-demand paradigm described previously in the section about network paradigms (2.3.3). sicsDAIS knows nothing about how to present agents, the agents possess this know-how, and this is what is transferred to the system in the form of content handlers. In a sense, sicsDAIS is a specialized “place” that handles interaction with the user at the “place”. Content handlers also have an interface to sicsDAIS much as Aglets have an interface to a “place”.

4.3 Analysis in respect to the demands

The demands that were mentioned in section 2.4 could be viewed as a minimum set for a system that is based on a dynamic agent environment. They are access, flexibility, dynamics, multiplicity, openness, and adaptivity. The question is how well the systems examined in the previous sections meet the demands.

- Access. The OAA allows users to access application agents (such as an email agent or a calendar agent) either directly or indirectly using the facilitator. Multimodal access is provided using special agents that handle the different input modalities. Aglets allow the user to access resources in a distributed environment by making the agents move from host to host in search for the required information. When an Aglet has performed its function, it returns to the user to present the results.
- Flexibility and dynamics. The OAA provides many means of user interaction built into the fabric of the architecture. Different agents are good at dealing with different modalities; there are for example agents for speech recognition and speech output; there are agents for gesture recognition and handwriting recognition. The user can interact through any agent’s

interface (that allows the modality in question that is) regardless if that agent can handle it or not. If the agent is unable to process the user's input the task is given to the facilitator, which will forward it to the appropriate agent. New agents for handling new modalities can be introduced into the system at any time. This results in great flexibility for the user. Application agents on the other hand must be woven into the application when the system is designed. For example, in the interface of the automated office application there are access points to a number of application agents such as the calendar and the email reader. If one wanted to add a conference reminder agent to the application one would have to rebuild the interface. In sicsDAIS, the ability to add new service agents (that use GUIs or not) is built into the system.

- Multiplicity. In OAA it is possible to interact with multiple agents simultaneously. This is especially apparent when a task is coordinated by the facilitator, which makes use of any agents available that can handle the subtasks. Numerous Aglets may be created to perform a task either for efficiency or for reasons of robustness. The user interaction with Aglets is based on the model of a separate mode of interaction for each Aglet.
- Openness. Application agents can be incorporated into an OAA application by wrapping them with an OAA agent wrapper. The wrapper isolates the application and provides the methods needed to cooperate with other agents in the system. The wrapping requires that the API of the application is available.
- Adaptivity. An OAA application may or may not be adaptive. The OAA architecture does provide the necessary structure for getting feedback for the user since the facilitator coordination between the involved agents can be monitored. As for adaptivity in individual application agents, this depends on the agent. If the agent makes use of any available feedback about the user, and the agent additionally is able to modify its interface or behavior, adaptivity may be possible. This is similar to sicsDAIS, where content handlers must make use of feedback to modify appearance and behavior. sicsDAIS only provides the framework for this.

In the Aglet Framework, interaction with the user takes place for each agent individually. There is no common interface in which interaction of several agents is coordinated with the user. This results in a complete freedom for agents to implement whatever interaction scheme they see fit, but it also looses the coherency of applications built using the framework.

In this and the second section of the thesis, we have discussed a number of different approaches to building and interacting with agent-based systems. There are varying ways of implementing agents and agent architectures and they each have strengths and weaknesses. This thesis focuses on the interaction aspects of such systems; how users interact with the applications that are built using the agent frameworks and architectures. There seems to be three major alternatives (reiterating from the first paragraphs of the thesis):

1. Each agent is responsible for its own interaction with the user. This means that there is no central component for coordinating the interaction of agents in a system of agents. This is the scheme used in the Aglet workbench.
2. All interaction takes place in one common interface. This interface allows agents to surface to interact directly with the user, or the user uses the interface as a springboard to interact indirectly with agents. This is the method used in the OAA.

3. A combination of the two. Agents are allowed their own interfaces in the shape of combinations of content handlers, but the interaction between user and all agents takes place in a common interface—sicsDAIS.

4.4 Design choices in sicsDAIS

In this section, we make some final comments on the design of sicsDAIS, especially in respect to the systems that were discussed above.

4.4.1 State, scripts, and listeners

There are more similarities between Aglets and content handlers. A content handler, like an Aglet, has a state that is readied when the content handler class has arrived in sicsDAIS. In the Aglet framework, this is done internally in the Aglet, while in sicsDAIS the content handler is scripted to achieve its states. This gives a greater flexibility. Some state information may be hard-coded in the content handler class by its author, some may be sent along with the content handler class when it is first instantiated, and some may be sent to the content handler later in the interaction session during run-time.

Aglet classes are built by sub-classing an abstract Aglet class. This class provides methods for cloning, dispatching, etc., and it defines methods that have to be implemented to specify the behavior of the Aglet. All behavior of an Aglet is thus built into the code that makes up the Aglet.

This is somewhat similar to the approach taken for content handlers in sicsDAIS. A content handler class author creates a content handler class by sub-classing either the `AtomicCH` or the `CompositeCH` classes. State information may be hard-coded into the content handler class, in the constructor or in an extra initialization (and finalization) method, or it may be set at run-time. State information for a content handler could be such attributes as colors, fonts and the layout. Another part of the state is the mapping of Java level events that the content handler can produce (MOUSE_PRESSED, KEY_PRESSED, etc) to sicsDAIS level events that the content handler author defines. sicsDAIS level events, as we have mentioned previously, are content handler author defined, and happen in a content handler with some behavior as a result. The behavior is specified by scripts of the same form as the description script (P/I-description) that describes the whole content handler. By setting up the mapping between Java and sicsDAIS level events, the content handler class author prepares the content handler for run-time alteration of its event behavior. In addition to setting up the mapping of the events, the author can also hard-code default behavior for sicsDAIS level events into the content handler by setting the scripts. At run-time the behavior may be set or changed by altering the script that the event executes when trigged. When a Java level event is produced, the content handler announces a corresponding sicsDAIS level event. The event is tied to some behavior in the form of a script, and that script is executed and the appropriate action is taken. See section 3.8 for details. Since the behavior is described by scripts that may be changed during run-time, the system is more flexible.

The listener concept of Aglets is also present in sicsDAIS although in a different form. The script that is sent to sicsDAIS as a description of the content handler also contains subscripts that are tied to events that may happen in sicsDAIS. The *behavior* that should result from these events is described in the scripts.

4.4.2 Messages

Messages may be sent between Aglets by creating and sending message objects. A message object contains the message type (any arbitrary string identifier that is matched in the receiving Aglet's message handler) and an optional extra parameter. To send a message to an Aglet you need a reference to the receiving Aglet's proxy. To send the message object the method `sendMessage` is called on the proxy. The message is then passed as an argument to the receiving Aglet's `handleMessage` method. It is up to the Aglet author to implement the appropriate behavior for this method. The method should return true if a message has been handled; otherwise, it should return false.

In addition to sending messages using the `sendMessage` method, one may also use the `sendFutureMessage` method. It takes the same arguments as the `sendMessage` method but instead of returning a Boolean, it returns an object of type `FutureMessage`. This object is used as a handle to later receive the result of the transmission. The sending object will block on the `FutureMessage` object for a specified length of time and then retrieve the result.

A message in sicsDAIS is in the form of a script, the same type of script that is used for describing the content handlers and for specifying the behavior of events. Each content handler class or component in sicsDAIS implements the `Scriptable` interface and has the same method for sending messages.

Messages are sent asynchronously between content handlers using the event handler component. The message to be sent is passed as an argument to the `sendMessage` method of the content handler. This method passes the message to the event handler's incoming mailbox, which queues and retains the order of the messages.

Another advantage of message passing which is not implemented in sicsDAIS, is that messages, as opposed to direct calls on a reference of an object, may be authenticated by some central component before they are handled.

We have in this section examined the major similarities and differences between sicsDAIS and two other systems. In the next section, we will discuss issues regarding the implementation of sicsDAIS in a real agent-based application—KIMSAC.

Chapter 5

Utilizing sicsDAIS in KIMSAC

The KIMSAC[21] project [9,10,11] aims to provide integrated, kiosk-based access to distributed multimedia (texts, pictures, sound, HTML pages, video-clips, video-conferencing and so on) services for the public. The chosen domain for the project is that of government services in the area of social welfare entitlements—Social Welfare Services (SWS), and employment services—Foras Aiseanna Saothair (FAS). sicsDAIS was developed as a component in the KIMSAC system.

5.1 Aims of KIMSAC

The information kiosks will be used by the general public, with consequent high demands on the usability of the systems. The usage of the kiosks must be as easy as using an ordinary pay phone, while still providing a variety of services and means of communication. Some aims of the KIMSAC project are

- The KIMSAC architecture must be open to allow for modifying of existing services and the dynamic introduction of new services. The domain data of the application services may change and this must be easily dealt with.
- The services provided, the social welfare domain and the unemployment domain, are complex domains. The rules for calculating benefits for example are dependent on both of the domain areas. The complexities of the domain information that the services provide, especially when the information is accessed by individuals, must be dealt with.

[21] KIMSAC: Kiosk -based Integrated Multimedia Service Access for Citizens

- The services provided must be adaptable to a variety of users, ranging from users with no prior computer or domain experience, to highly experienced users. The kiosks must also serve users with varied sorts of disabilities such as physical handicaps or dyslectic problems.

To face the challenges mentioned above, KIMSAC is based on an agent approach to the system design. Agent technology is used at two levels. On the interaction design level, an interface metaphor (which in KIMSAC is called the Personal Service Assistant (PSA)) is used to realize user adaptive assistance (more on this below). On the architectural level, the agent design metaphor is used for dealing with complex functionality and realizing an open service environment.

Two general categories of service agents exist within the project: *domain service agents* and *personal service assistant agents* (PSA agents). PSA agents are personal to a user, and are used to tailor access to the service agents for the user's individual needs. Domain service agents implement a number of facilities provided by a service agency, the typical examples being the social welfare services represented as one agent and employment services as another. These were studied as an initial application domain for KIMSAC's user trials.

sicsDAIS is a central part of the KIMSAC architecture. It provides the interaction interface between the user and the services of the system. Note that sicsDAIS is not specifically designed to handle the PSA's interaction with the user, but rather as a general tool for presentation and interaction that the PSA makes use of.

5.1.1 Architecture

The KIMSAC architecture identifies five major components (Figure 9). These components can be summarized as

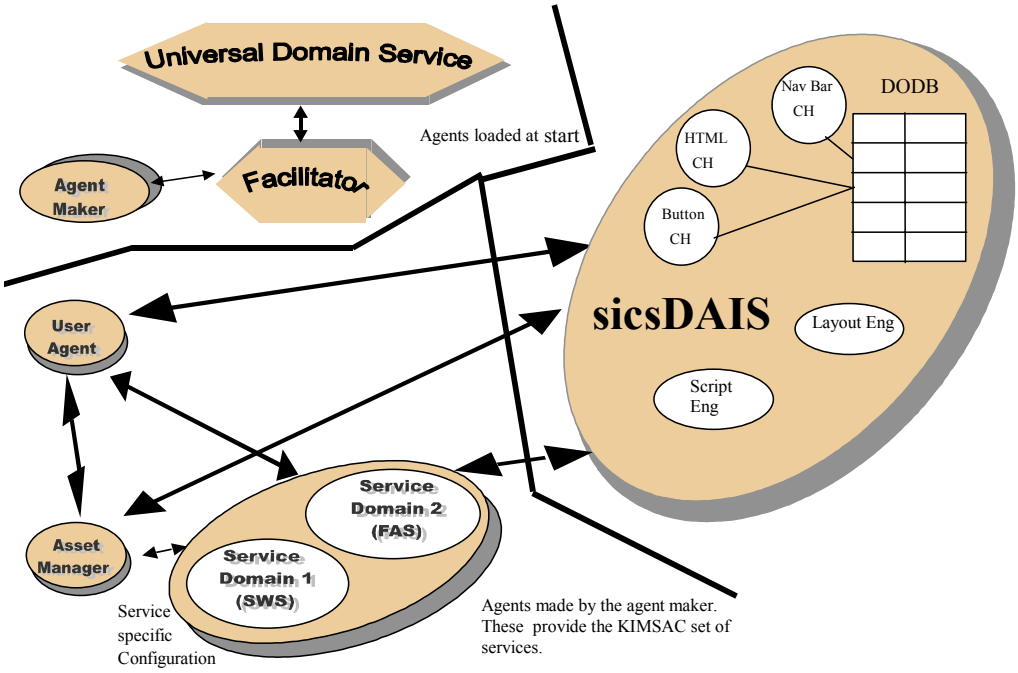

Figure 9. The architecture of KIMSAC

- SICS Dynamic Agent Interaction System (sicsDAIS). Presentation and interaction system for multimedia content rendering and management of multimedia content (screen layout) and interaction (as has been described in this thesis).
- Multi-agent architecture for high level reasoning. These are service agents and agents supporting the PSA functionality.
- Distributed service management platform for heterogeneous systems using Agent Services Layer (ASL) (across platforms and language technologies) and general service access to legacy databases, security and system management. The ASL provides a distributed and open platform for deploying agents that communicate using KQML messages. It does not impose the use of a particular content language. ASL is based on the concept of the *authority*, a controlling agent and name server, which manages the creation and deletions of agents within its domain [45].
- Distributed ontological database to support multimedia content re-use and agent access to multimedia content.
- Rich communication language to share information and service access at a flexible level, thus permitting both sub-components and major components to be integrated in a loosely coupled manner (KQML and KIF).

5.1.2 The Personal Service Assistant

The personal service assistant metaphor is used to increase the usefulness of the services and the access to the services. It depends on the user's goals and tasks in modelling the interaction

between the service agents and the user. The requirement of the PSA is thus to explicitly model the users goals and tasks.

A PSA also has to take into account the services' perspectives on the user interaction. Specifically, a PSA is required to provide help in the following areas:

- The kiosk and how it works
- The different services provided and cross-references where appropriate
- A given service and its applicability to the user

Furthermore, a PSA tailors help to important user characteristics, such as

- To the expertise level of the user in using the kiosk or any given service
- With respect to the user's situation
- To the general cognitive capabilities of the user

The PSA metaphor can be viewed from three broad perspectives:

- The *personal* aspects of interaction—meeting the client's specific and personal needs
- *Service* access—being service centred, providing access to services the user may not be aware of
- *Assistant*—co-ordination, filtering and extending information and service access

Essentially a PSA requires the same set of components as a service agent. However, the PSA is not a specialised service agent, such as a news information provider but is a combined application of the user requirements and the services available. For the PSA to interact with a service provider the requirements are a communication protocol, a communication channel and a method of knowing where and how to contact that service. However, the PSA also needs to be able to support the user in using the system and therefore needs a computational model of the system interaction mechanism.

5.1.3 Inside sicsDAIS

The sicsDAIS system is used as the interface to the user in KIMSAC and allows agents to present information and to interact with the user. In section 3, we discussed content handlers in general terms. In the KIMSAC system, sicsDAIS loads and coordinates *application-specific* content handlers, some of which stay in sicsDAIS for the length of a session. They are non-visual content handlers (they have no user interface) that handle coordination and adaptivity in sicsDAIS during the course of running KIMSAC. They are the Task Manager (TM), created by Olle Olsson and the Adaptive Agent[22] (AA) by Markus Bylund.

The TM manages other content handlers that are active in sicsDAIS during a session. It coordinates the communication between agents and content handlers by modifying the content language from some agents to suit the content handlers. It supports the sharing of data, it supplies the domain object database with data that it has stored internally (it can thereby simulate outside agents by entering data in the database in areas where an agent would be

[22] The name Adaptive Agent suggests that this content handler is in fact also an agent. In the KIMSAC system, this is not technically the case since the AA does not communicate as a peer on the ASL. From a more philosophical point of view, the AA is indeed an agent.

expected to enter data), and it creates supporting content handlers when they are needed. Thus, the service agents responsible for the instantiation of the content handlers need not be involved in the details of the content handlers execution once they have been created. It also means that the TM is able to ensure that new services that are introduced in the interface do not completely alter the behavior of the interface. The TM can release ("kill") existing content handlers and create new content handlers on command from the service agents.

The main reason for placing the TM inside sicsDAIS, instead of making it an agent, peer to the service agents, is that a content handler has a greater degree of access to the internal components of sicsDAIS than does a service agent outside. An internal agent can monitor and modify the communication between content handlers as well as between agents and content handlers. It also has nearby access to the domain object database. The TM can thus more effectively coordinate the work of the other content handlers using the database[23]. The use of the AA as a content handler follows from the same rationale.

The AA is responsible for adapting help content in three categories, instructional texts and video, help about where to go next, and a glossary. The agent will only adapt the choice of what help to provide and not the contents of the help itself.

Figure 10 shows the user interface at a point in a session when the user is building a "user profile" by supplying specific information (age, marital status, children etc.) which will allow

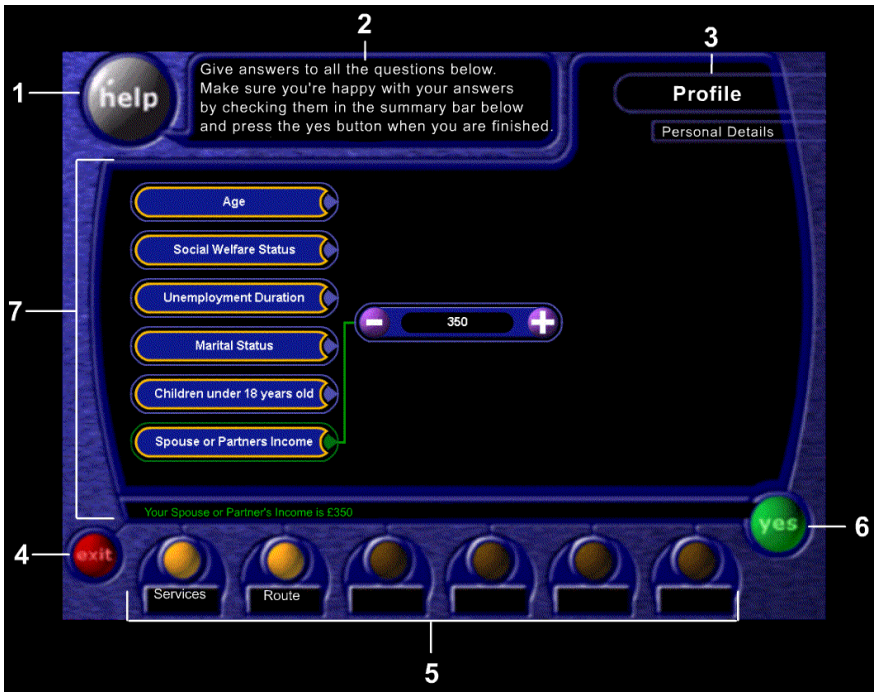

Figure 10. The KIMSAC user interface showing instantiated objects as different content handlers in sicsDAIS.

[23] After using the TM as part of the KIMSAC system, we have concluded that a subset of the TM functionality—the ability to spawn content handlers internally, the ability to enter data in the domain object database according to some predefined state machine, the ability to filter content language—will be useful as a built-in component of sicsDAIS.

the agents to provide focussed information on income support, training and job availability.

The AA controls the instruction and information area (number 2 in the figure) of the KIMSAC interface in sicsDAIS as well as the help button (number 1) which supports the video representation of the PSA. The instruction text and video provide help for the user's immediate task. The help is usually limited to the information that is presently displayed on the screen. The help texts and the video clips are created prior to the start of the session, as is the mapping of the system state to the applicable help content. The adaptation is done by analyzing the user's history with the system, which is kept in an interaction profile. The profile together with the state information maps to the user's need for help in terms of three levels, complete, brief, and none. The stereotypes are associated to different system states, which allows for different help levels in different situations. The user can also enforce a particular level at any time [5].

The glossary is a small table of terms with associated explanations. The table is compiled at run-time by the AA using explanations provided by the service agents. The choice of which terms to include is also based on information provided by the service agents. The adaptation of the glossary lies in the choices made about the included terms, which is dependent on the system state.

In summary, the elements of the interface are

1. The help button, which supports the video representation of the PSA, is supported both by sicsDAIS and the AA. At each screen-level interaction, the PSA is visualized providing spoken information. This is one level of navigational help that is supported during interaction. At any point during the interaction, the user can request help. Further help and explanations are supported by the AA at this interaction level. The user can set his or her level of expertise for this particular service access.
2. The instruction and information area is managed by the AA, which holds an interaction profile model of the user to assist in providing the correct level of instruction and help for the user.
3. Current task information is supplied to sicsDAIS via the service agents. It is the TM within sicsDAIS that co-ordinates this information.
4. The exit button communicates with sicsDAIS to close down the system and instruct the agents that service requests are now terminated.
5. The navigation bar communicates with the TM in sicsDAIS. Each button is supported by a particular agent. For example, the route button is supported by the current service agent. When a navigation button is pressed then an action is invoked to inform the correct agent.
6. The "yes" button is under the control of the agent presently providing information.
7. This work area is where service information and access is provided.

The user does not need to be aware that there are a number of agents acting on their behalf. For the user, there are two central points: the visual PSA for screen to screen interaction and the service area. Other information is either passive to an extent (instruction text) or requires the user to interact proactively (e.g. navigation bar and help). Control of the interaction is within the user's control. However, management of tasks and task information at complex levels is handled by the agents. The KIMSAC system, operating as a PSA, ensures that the user is guided through the system in an unobtrusive manner. This is done via the protocols set up between the agents supporting the PSA and the services:

- The PSA controls the back-end information at any general level that is client specific (identification, help level requirements, user name, etc.)
- Control is passed to the expert service agents only when a client is ready to pass control (selection of service).
- Centralized adaptation of information occurs at specific points in a consistent manner (links screen, related jobs and social benefits on service screen).
- Coordination of information takes place within a service and is presented to the user via the results screen.
- Coordination across services in an effort to achieve client-centered information is done through a cooperative protocol between the PSA and the service agents.

5.2 Contributions of sicsDAIS in KIMSAC

What are the demands of sicsDAIS as a component in the KIMSAC architecture in particular and as an interaction system in an agent based information system in general?

5.2.1 Communication

sicsDAIS is required to be a fully functional member of the communications infrastructure of the agent system in order to provide the services of presentation and interaction to the user and the agents. In the case of the KIMSAC architecture, this means being a component connected via the ASL. The system can thus receive requests from, as well as communicate interaction requests to, other agents on the ASL using KQML.

sicsDAIS must also enable content handlers to communicate internally. One part of this communication is the actual sending of messages using the event handler. Another is the provision of a shared data storage facility. We have discussed the demands in sections 2.4 and 3.9.

5.2.2 Presentation and interaction management

The two main objectives of sicsDAIS in KIMSAC are

1. To present various types of information and multi-media content. Moreover, to do so in cooperation with service agents that provide the content and coordinate and define the presentation/interaction sequence.
2. To provide the user with an interface that enables direct or indirect interaction with the agents providing the information.

For this, sicsDAIS needs to have flexible layout management. It must be able to render any type of content regardless of modality and even types of content that are unforeseen at the time. It must coordinate information on the screen for the user and interpret the correct action to be executed when a user interacts with the system. This will, in KIMSAC specifically, enable the presentation of the PSA in whichever form is most appropriate at the time. It may mean an anthropomorphic and proactive entity in the actual interface, or more subtly, presentation traces of the PSA as the elements of the presentation and interaction sequence are controlled and scripted by the PSA.

Concretely, sicsDAIS must understand the content language being used: in KIMSAC this is the P/I-descriptions. These representations of presentation/interaction sequences must be processed and translated into real useable interactive elements in the user interface.

5.2.3 Open service provision

Related to the aspect of dynamic and flexible presentation is the matter of open service provision. Not only must sicsDAIS be able to incorporate new types of content as mentioned above, but it should also promote the reuse of the elements of the presentation and interaction. These elements should be constructed in a way that enables the assembly of simple elements into more complex ones. This gives the greatest degree of flexibility and reuse.

New content handler classes can be installed dynamically into sicsDAIS during the course of running the KIMSAC system, and new, more complex content handlers can be assembled from other simpler ones.

5.2.4 Implementation

sicsDAIS is a platform for multi-media content handlers, but a complete presentation/interaction sequence or task may need more complexity closer to the user than what the content handlers themselves may need. Therefore, sicsDAIS must be able to accommodate content handlers that can work behind the scenes of a presentation, and yet be close to the interaction occurring in the interface. This is possible using invisible content handlers (e.g. the TM and the AA). These may react quickly to communication taking place between other content handlers in the interface or they may coordinate the activities concerning the domain object database. For example, an invisible content handler can be loaded into sicsDAIS at the start of a presentation/interaction session. It can contain data for inserting into the domain object database at appropriate times, data that will be used by subsequently created content handlers. In essence, the whole presentation or interaction sequence can be scripted in this way, being completely isolated from interacting with any outside service agents.

On a more practical note, sicsDAIS must be able to run on multiple platforms for portability and availability. It must also allow separate threads of execution to allow for easy installation and de-installation of the independently running content handlers.

As one can see, the demands of the KIMSAC system of the system of interaction have served to shape the design of sicsDAIS. The system has thus become an integral part of the KIMSAC system while remaining unspecialized and flexible enough to fit as the interaction system in a more general agent based information system. Flexible layout management combined with the capability for content reuse and the ability to dynamically render P/I-descriptions all support the seamless introduction of new service domains.

Chapter 6

Future work

sicsDAIS is a platform-independent system for user interaction with agents. It is a central point for interacting with multiple diverse agents that all use content handlers for their presentations and interaction with the user. sicsDAIS is the vehicle for this interaction in that it provides the means for the interaction to take place, without itself having any logical impact on the presentation. The system is open and flexible in such a way that new agents can be introduced at run-time by dynamic loading of the agents' content handler classes, the parts of the agents that handle the interaction and presentation. sicsDAIS is useful as a general interaction/presentation component in an agent-based application environment, particularly if there are high demands on flexibility and portability. In this, the final section of the thesis, we will discuss possible applications of sicsDAIS that will demonstrate its wide range of uses, as well as more detailed improvements that can and will be made, and finally a look towards the future.

6.1 sicsDAIS in an open agent architecture

ConCall[24] is an agent-based system for collecting, filtering and browsing of conference calls. Using ConCall, an individual researcher can review calls and set up reminders for deadlines. To avoid uninteresting calls, the user can set up a filter to retrieve a personal selection of calls and organize them in a personal manner.

ConCall is built on the principle of bringing the human into the loop; human editors take part in the filtering and classification process of the domain data. In the case of ConCall, editors classify all incoming conference calls using 'buzzwords'. A buzzword is similar to a keyword

[24] ConCall is an application of the Edinfo [36] agent services architecture.

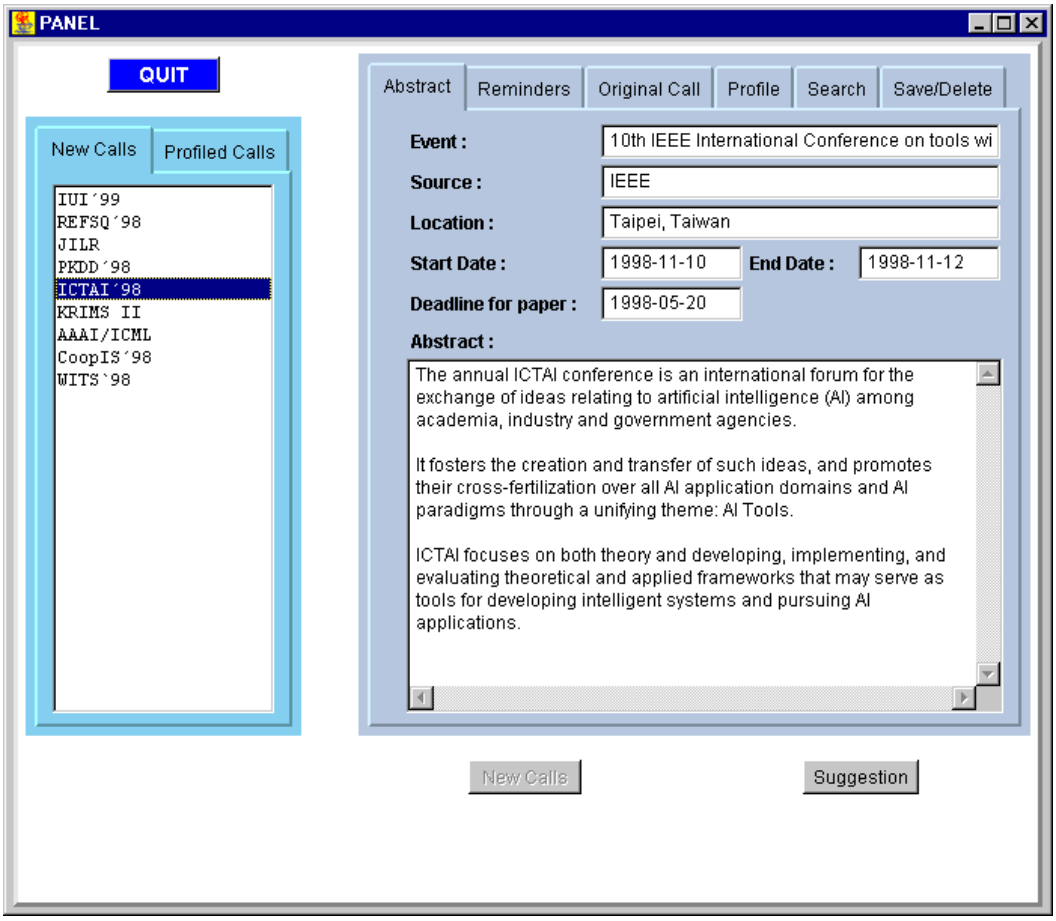

Figure 11. The ConCall interface.

except it is not tied to any formal or global ontology or agreed upon set of words that may be used. A 'buzzword' is just as likely to surface into the system originating from a user as from the editor. The purpose of using 'buzzwords' instead of keywords is to allow for a more flexible and self-adjusting body of classification words. The editor uses the 'buzzwords' to classify calls, which are then filtered according to the preferences of the users. Users may also set up their own set of filtering rules using their own 'buzzwords' for a second level of filtering or ordering. These user-defined words are provided as feedback to the editor who can react to trends or new topics of interest by incorporating the new words.

The ConCall architecture

The ConCall service is built on an agent architecture, which consist of a number of specialized agents. These are

- The service assistant. This agent handles the interaction with the user and provides a central point of interaction between the user and the agents in the architecture.
- The profile agent. This agent stores the preferences of the user. This profile is based on information about the user's actions that it receives from the ConCall agent. The user can inspect and change the profile.
- The ConCall agent itself. This agent does the filtering of calls for each user.

- A reminder agent. This agent provides a reminder service to the user and to other agents. The user may be interested in getting notification prior to a deadline for submission of papers to a conference, and the reminder agent will handle this by sending an SMS[25] or email to the user at the appropriate time.
- The database agent. This agent handles transactions with a database that stores the conference calls. Calls are entered into the database by the editor.[26]
- The logging agent. This agent is accessible from all other agents and enables agents to keep a record of events.

The user interaction

In the first implementation, the user interacts with the ConCall system using the service assistant agent. This is a Java application that has a number of tabs, one for each service that may be reached in the system (see Figure 11). The service assistant is hard-coded to interact with the ConCall agent, the reminder agent, and the profile agent. The user can switch between interacting with each agent by selecting the tabbed screen that corresponds to each agent. In the interface, there are two tabs that are directly mapped to agents (reminders and profile) and two for viewing the calls (abstract and original call). There is no tab for the ConCall agent as the interaction with this agent is done through various elements in the interface.

There are a number of problems with the implementation of the service assistant. Firstly, the agent is hard-coded to handle interaction with the agents mentioned above. If a new service is introduced, the service assistant will have to be rewritten to be able to handle the interaction with the new agent. Secondly, it is difficult to introduce third party agents and services into the system. Any new agent will most often require some means of interacting with the user, which in turn requires a rewrite of the service assistant. This is possible when creating new services within the ConCall project, but impossible for third party vendors that do not have access to the source code of the service assistant. Thirdly, the interaction form in terms of what data is exchanged between the user and the agents, as well as the layout of the interface, have to be set at the time of writing the service assistant. For one, this means that the interaction is set at the time of creating the service assistant. The service agent can only provide the data that was foreseen at the initiation of the new service and no changes can be made thereafter. Also, there can be no changes in the *services* of a service-providing agent since that also requires changes to the service assistant.

Using sicsDAIS in ConCall

A possible solution to the problems described above is to use sicsDAIS as the user interface of the ConCall system. In a way, sicsDAIS would take the place or rather *become* the service

---

[25] Short Message Service. A messaging service provided in European GSM mobile phone networks.

[26] Note that this architecture is a prototype version where the editor is not explicitly represented in the architecture. Calls are collected, classified and entered into the database using tools available to the editor that are not part of the users' collection of agents. In a forthcoming implementation, the two will be integrated.

assistant. The interface could be made to look the way it does in the prototype, but with added flexibility:

- No functionality for interaction with the user is hard-coded into sicsDAIS. This means that new services can be introduced at any time.
- It is not a problem to incorporate third-party services. The only requirement on their part is that they build content handler classes for the interaction.
- The patterns of interaction between agents and the user do not have to be specified prior to running the system. Changes to the layout, form of data, modalities, and services can be made on the part of the service providing agents as the system is running. Any new or updated forms of interaction are automatically and dynamically loaded into sicsDAIS using content handlers.

As we can see, it would clearly be advantageous to use sicsDAIS in a setting such as the ConCall system. However, what exactly are the characteristics of the ConCall system that make it such a fine candidate? There are requirements of the ConCall system that may be typical of many agent-based applications in an open architecture:

- Users wish to utilize the services of agents.
- Agents require interaction with users.
- The system has to be open so that new agents and services can be introduced.
- Services should be able to change dynamically in terms of the information provided and the format of the interaction of the agent with the user.

These requirements may be provided for by sicsDAIS.

6.2 sicsDAIS in small devices

Let us now examine a different setting, that of mobile computing. In this situation a user is equipped with a mobile computing device, such as a laptop computer, a palmtop such as the PalmPilot [63], a Windows CE device [74], or a mobile phone. The common denominator between these types of devices is that they are portable and yet have access to the network. They also have limited input-, output- and computing capabilities in comparison to desktop computers.

Advantages of mobile computing devices

Applications such as calendars, contact organizers, and lighter versions of desktop applications can be brought along to be used anywhere. In combination with a wireless network connection, this enables the user to access email and the World Wide Web along with for example the corporate intranet. It also allows access to agent-based applications in the user's usual working environment. This means that the same services and applications that are used at the desktop can be used on the road.

Constraints of mobile devices

However, a mobile computing device is different from the usual desktop computer or network terminal. Although smaller mobile devices like palmtops and mobile phones usually have output means such as screens and speakers, these are usually much smaller than their desktop counterparts. Mobile devices, because of their size and battery capacity, usually also have

limited processing power. Finally, the input capabilities are sometimes very constrained, spanning from a minute keyboard, to character input using a stylus, to no text input at all but rather speech input as in a mobile phone.

The restrictions inherent to the small devices are mirrored in the applications. The light versions of the user's ordinary desktop applications are usually constructed to better suit the smaller devices in terms of smaller screen real estate. There are also fewer and less powerful functions available because of the usually moderate computing power of the mobile device. The same constraints apply to the interaction with agent-based systems. The agents must be able to adjust their interaction format to suit the device, in terms of size, modality and bandwidth. If an agent needs to execute on the device, similarly to the lighter versions of the desktop applications, it will have to be modified in terms of processing needs to run on the device. This may be difficult since agents usually provide some service that is associated to resources that exist on servers in the network. The agent instead has to present its interface to the user and the interface rather, must be modified to work on the mobile device.

Interacting with agents using a mobile device

In this scenario, sicsDAIS will ease the transition from the desktop to the mobile device. The system is lightweight in terms of processing needs since it contains little functionality in itself. The agents run on central servers and only supply content handlers to sicsDAIS. The system is also designed to run on multiple platforms, the same platforms that support Java. If the mobile device supports Java, it also supports sicsDAIS.

Regarding the use of the same agents and services in the mobile environments as the desktop, there are two possible solutions: (1) sicsDAIS handles the formatting of content handlers to suit the environment (this is being done to an extent, see section 6.3 and *the layout algorithms*, below), (2) content handlers are sensitive to the environment.

In the second case, agents will have to keep the content handlers flexible. Content handler classes have to contain code which will sense the type of environment in which sicsDAIS is running. They can then adjust parameters such as method of presentation, the layout, or the modality to the current device. There is no need to have specific modes of interaction for specific devices, but rather to have interfaces that can adapt gradually to the environment. There are two alternatives for implementing this on the agent side. In the first, a special agent that is capable of sensing the constraints of the interface as a whole coordinates the interaction. This agent can mediate the interaction between the user and the other agents. The other alternative is to encourage agent developers that use sicsDAIS for interaction to build adaptable content handlers.

A combination of the two techniques will most likely yield the best results. Content handlers should prepare for differing environments by including methods for interaction using different modalities and layout schemes, while sicsDAIS will provide the sensory information. There will of course be a balance in sicsDAIS between the strict conformity to specified layout and the flexibility of a lax interpretation (e.g. the rendering of web pages in a web browser).

Mobile computing is set to take computing to new levels of usefulness. This will mean new challenges to the design of user interfaces. sicsDAIS' place in this development is to provide a unified platform for users to interact with agents, regardless if the environment is the desktop or a mobile device. In future projects, we will experiment with using sicsDAIS as the interface to our existing agent-based applications.

6.3 Specific improvements that will be made

There will be a new version of sicsDAIS in which a number of things are going to change. Following is a short overview of some of the possible changes or additions that may occur.

The layout algorithms

Using the provided layout algorithms of sicsDAIS works well in most cases when the author is not concerned with the exact positioning and sizing of components. The layout algorithms need more work when it comes to intelligently sizing and placing components, especially in certain circumstances. If the number of components or the combined size of the components inside a composite content handler becomes too great, they simply will not fit.

The goal of this effort is to be able to provide a sicsDAIS that intelligently, with the help of preferences of the displayed content handlers, can do an adequate presentation in a greater variety of environments including mobile devices.

Portability of sicsDAIS

Currently, sicsDAIS is a standalone Java application that must be installed on the computer where it will run. In the future, we will modify the system so that it will be possible to access it in a transient manner. One way to achieve this is to implement it as a Java applet. It could then be downloaded to the user over the network. This is a good idea if the service that is provided using sicsDAIS is of the type that needs to be accessed expeditiously without the user having to bother with installing new software.

Caching of data

A typical sicsDAIS presentation and interaction session is made up of a series of P/I-descriptions with related data. These are displayed using content handlers. Content handlers are often generic enough to display different data depending on the situation. The data is sometimes in the form of text, and other times in the form of pictures, audio, or video clips. It is often stored in a library at the agent site. This means that the P/I-descriptions and the data have to be mirrored at the site of sicsDAIS or downloaded at presentation time from the agent host. This downloading of data results in a great deal of network activity, and to alleviate this, a scheme for caching (temporarily saving data locally for later reuse) could be used.

sicsDAIS will have to be redesigned to include a caching handling component that can temporarily store used P/I-descriptions and media. As the data grows older, it will disappear from the cache. For security reasons, care has to be taken to prevent content handlers from using each others data. Another possible solution is to integrate the caching in the domain object database.

Library of content handler classes

There is a need for a library of simple generic content handler classes that may be used by all agents for simple interaction and presentation sequences. For example, a simple dialog box asking for confirmation could be used in a number of circumstances. If these generic content handler classes are designed to be completely configurable, they could be easily reused. The functionality and parameters of each such content handler class would have to be clearly

documented and made available to interaction authors. The content handler classes themselves would be provided in a JAR[27] file accompanying the sicsDAIS distribution.

An additional advantage in using generic content handler classes is that such classes would not have to be downloaded to the client sicsDAIS at the time of presentation. This will further contribute to minimizing download time for users.

Mapping between content handlers and JavaBeans

There are many similarities between JavaBeans and content handlers as we have observed in sections 2.2.2 and 3.5. It would perhaps be useful to provide a mechanism in the base content handler class for incorporating a JavaBean as the main interactive/presentation unit. This would mean that a JavaBean could be developed for some purpose and then be reused as a content handler class in sicsDAIS. To achieve this a special kind of content handler class that would serve as a wrapper around the JavaBean would have to be created.

6.4 The future

This last section of the thesis is a look ahead to the future. I will describe some of the ideas that have surfaced during the course of working with sicsDAIS, ideas concerning user interaction with future agent-based systems, which place the agent interaction system in a much greater context. I will first speak of the situation such as it is today and then I will offer what seems to me like a plausible scenario for the future. This scenario describes the role of a more advanced sicsDAIS component in an agent-based architecture. However, it is also very broad and therefore includes many related aspects of user-agent interaction that seem to be important.

6.4.1 The problem

The problem is the web. The web has suddenly caused a massive increase in the amount and variety of information that ordinary people have accessible. This in itself is not a problem, but rather one of the more profound ingredients for technological and sociological change in the history of mankind. The total access to global communication and the sharing of all information will change the society in which people work, play, and socialize. The web in itself is not the key to this transition; it only acts as a crude medium and as a first catalyst to the transitional process. The bodies of information and the emergent effects of coalescing them are the important factors. The web is the problem because it has already outgrown its usefulness—it has turned into a big bulletin board where everyone advertises and sells products. There is useful information in this *infosphere*, but it is not exactly readily available.

The problem is not the huge amounts of information. We need all the information we can get to achieve the smelting pot of spontaneous information and service clustering. The problem right now is that we cannot access the information because the web medium is not powerful enough. More bandwidth will not make a difference, nor will more powerful search engines (unless of course they are used to search for data that has been classified as to content). What is missing are new ways of interacting with the information. We need new tools and new combinations of the tools for this. Take for instance Shneiderman's ideas about visualizing data or Maes' about delegation. These may be essential ingredients in a new *Open Agent World*.

---

[27] Java Archive [38]

sicsDAIS and the agent interaction model

What then is the first stepping stone towards a solution? In the work described in this thesis, we have made a first attempt at trying to concretize one solution to the difficulty of interacting with agents. sicsDAIS is a platform for experimenting with new models of user-agent interaction. The KIMSAC system, from this perspective, is an instance of a model of such user-agent interaction. The model can be described as follows.

Given a set of service agents and the goal of providing a combined application based on the available services, how does a user interact with the application? The model is based on providing a common interface (sicsDAIS) where the user can interact with all the agents using their *content handlers*. There is some degree of interdependence between the agents, in sharing information about the user and in the handling of the sequencing of interaction steps by a manager (the Task Manager). The agents are separate entities, but appear to the user as one application.

The model works well for a system such as KIMSAC where the service agents are developed within a consortium in which partners have some knowledge of the requirements of each service agent as a part of the whole. Much of the glue of the application is provided implicitly in the P/I-descriptions that the agents use to present information as well as by the Task Manager. The weakness of the model is in extending the range of services. This does not mean that there are no provisions made for introducing new services or for incorporating these with existing services, but this is not enough. If we do introduce a new service agent it will be accepted into the general agent architecture and it will be able to interact with the user through sicsDAIS. However, there must also be a way for a user to reach new services without requiring possibly independent service agents to be dependent and to know something about each other.

In KIMSAC, the service agents are presented in the starting screen of a session, as a number of buttons. This screen is a menu of possible paths to take in using the system and the new service can be added here. The reason we cannot add the service to any arbitrary point in the sequence of the user's interaction, is that this would require us to somehow make some existing agent aware of this new service. That agent would have to include some type of reference to the new service in one of its interaction situations.

The problem with an open agent architecture

We are faced with a contradiction. On one hand, we wish to keep the system open to allow new services to be added transparently. We do not want to place constraints on agents in terms of knowledge they need to have of other agents. It should be possible to create a new service agent completely independently of other service creators. On the other hand, we would like to see some type of cooperation between agents, such as sharing of user data or agents servicing each other as a means to complete the user's tasks. We will approach the above problems from the user's perspective—how will a user handle the information and service processing tasks in the future.

6.4.2 The vision

The vision is an *Open Agent World*. A world in which human users and software agents work together in order to get the job done. Both users and agents provide and consume services, some as end users and others as providers or refiners of information. Everything is connected. Information that is of no use to one part may be of great value to another. The Open Agent

World will connect suppliers and consumers with each other, while achieving efficient logistics of information and services.

Agents will trickle into niches of service where they can be of use. It will be an evolutionary process. Useful agents will survive (stay online and be available for service) and less useful agents will become extinct (fall into oblivion). The selection will be done by human and non-human agent users in finding the fittest agents. The usefulness will be defined by commercial success, numbers of users, trends, and fulfillment of purpose. New agents will appear as the result of cross breeding the ideas of older agent generations (or as products of real evolution in a concrete agent biosphere). Agents will become ubiquitous in the computer networks as they carry out more and more tasks.

Analogously to the web, this network of interconnected agents and users will provide information and services. A non-exhaustive list of services includes information access and refinement, electronic commerce, entertainment, and collaboration. Users will be able to book airline tickets, shop groceries, collaborate in virtual meetings, and access real-time business information, with much greater ease than today. Agents will handle tasks such as continuously monitoring information sources for relevant information, make purchases and pay fees, and collaborate with other agents to determine the estimated value of services.

How will this differ from the web? The main advantage to the web is the potential of delegation of tasks to the agents and the collaboration and communication between agents in performing the tasks (web pages don't talk to each other). A multitude of agents in various locations will be involved with or without the user's active participation. Agents that have been authorized by the user will sometimes act autonomously to perform tasks that involve constant attention or repetitive actions. Sometimes, while performing tasks with choices that must be made by a human, they will be under the direct supervision of the user. The integration of agents, resources, and human agents will be transparent to the user, who will only see the results of the efforts to accomplish the tasks.

The user interaction

The user will interact with the agent world using an agent browser, or rather, a Dynamic Agent Interaction System (DAIS)(see Figure 12). He or she will be able to access known and unknown, as well as locally and remotely situated agents, in the same manner. The interaction will primarily be graphically oriented with complementing modalities including speech. The system will be accessible in various forms depending on the computing device of choice.

The user is faced with a workspace with several windows for controlling and interacting with the agents (in my fictitious demonstrator system). It shows the visual representations of the agents and not the agents themselves, since these operate on the agent servers.

- One window houses agents that have been assigned tasks to perform (at the top in the Figure 12). These agents may be roaming the network searching for resources or they may be executing on a remote server. The window is used to communicate with the agents and to monitor their activity.
- The second window is a collection of minimized active agents (at the bottom in the Figure 12). Any active agent may be placed here temporarily.
- Another window displays a list of currently known agents (on the right in Figure 12). In this list, agents are identified according to the services provided, etc. When new agents are discovered or introduced they are listed here.

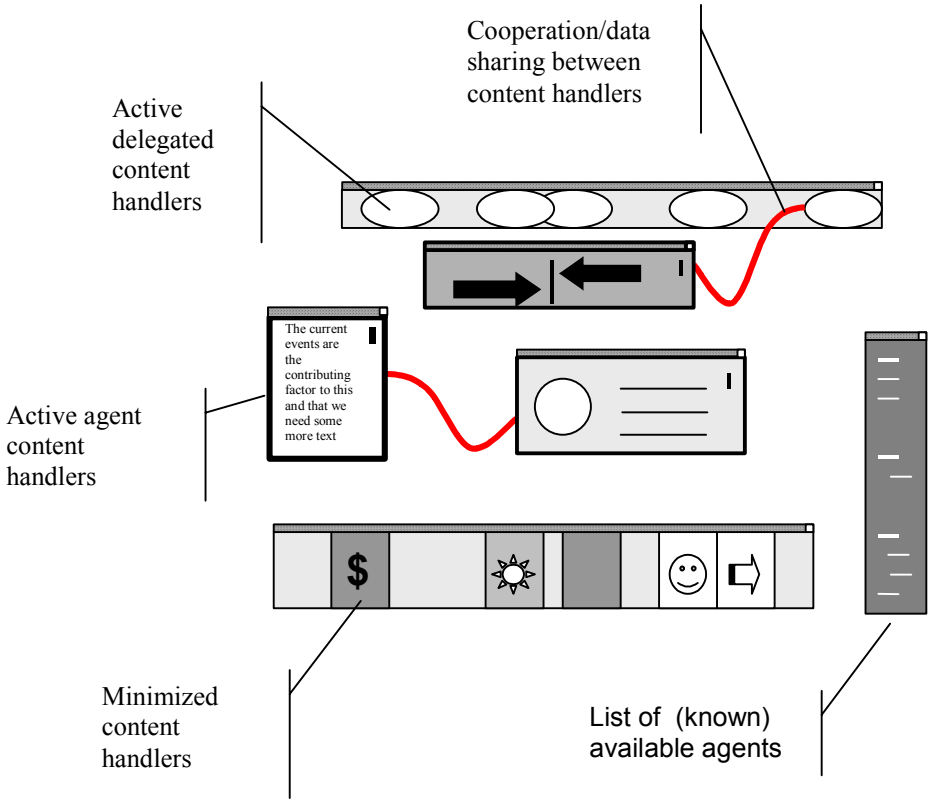

Figure 12. An agent interaction system for the Open Agent World.

- All other windows are individual active agents. Each agent is assigned a window in which it may present itself and interact with the user. Agent visualizations in the interface may be moved, minimized, deployed to perform tasks or deactivated. The user can zoom in on an agent or keep several agents accessible simultaneously.

As agents are created (the content handlers of the agents, not the actual agents) in the interface, they will announce the services or data that they provide to the interaction system. They will also be allowed to access the common pool of services and data that is made available to them in the system. This pool contains descriptions of the services and data offered by the active agents and the known agents in the known agents list, as well as data concerning the user.

Any usage of services or data available in the pool is subject to access restrictions. Some of the data will be freely available and some will be restricted. Some agents will automatically be allowed access by showing proof of some degree of trust or fulfillment of clan belonging. Others may be allowed access manually by the user, after he or she has taken the necessary precautions to ensure their benign intent. This is an important point. The use of the data and services should be as automatic as possible in cases when access has been approved, but it should also be possible for the user to manually control the access at any time.

Services provided by active agents may be used at any time. Services provided by agents in the known agents lists are not readily available until the agents are made active.

The user data will include information about the user; preferences for the presentation, security access information, etc. This data may also be administered by local personal assistant

agents that handle the user model, the user's preferences, the user's security signatures for access to services, etc.

A usage scenario

Let us examine a simple example situation in which the user will discover, access, deploy, and monitor a number of agents (the example is chosen for simplicity, not necessarily for typicalness. Many other usage scenarios are plausible and perhaps even more probable).

The user is equipped with a Dynamic Agent Interaction System (DAIS) as described above, that happens to have the following agents active at the start of our demonstration

- A personal assistant containing a user model
- A bank agent
- A search agent

Let us assume that the user wishes to travel to Hawaii on vacation. He will start by deploying the search agent to search for tourist agents from Hawaii. While this search is in progress, he will interact with his bank agent to find out if he has sufficient funds. The total funds available are calculated by taking the current balance, adjusting it in regard to this month's bills (some are known and some are estimated from past months), and adding some of the credit that the user has available. While making assumptions about the user's habits and preferences the bank agent communicates with the personal assistant. The bank agent has previously been authorized to access this information. At certain times, it also confirms assumptions by querying the user. Finally it presents a total estimate, and the user has to decide if he can afford the trip based on the result of the calculations.

At this point, the search agent signals that it has found a number of tourist information agents as well as travel agency agents specializing in Hawaii. The user is somewhat overwhelmed by the results and decides to filter out most of them. He therefore directs the search agent to query any opinion/review agents of which it is aware. The number of hits is quickly reduced to two. Any previously unknown agents (the opinion/review agents) are added to the list of known agents.

The user starts browsing the information available about Hawaii and finally decides that Maui is the best place to go. On command from the user, the tourist agent relays a query about flight information to the travel agent. There are a number of flights available but they seem very expensive. The user connects the tourist agent to the search agent (this is shown in Figure 12 as a curved line between two agents) and specifies a search for cheap tickets. The result is an auction agent, a courier agent, and a few budget travel agents. One of the budget agents is cheap enough and the user connects this agent to the bank agent, which transparently transfers the funds after getting authorization from the user. An electronic ticket is issued and sent from the travel agent to the users personal assistant.

You will notice that the user is accessing the interaction system using a combination of techniques. Agents are configured and monitored using direct manipulation, while search tasks are delegated. The user is accessing the interaction system as a whole using direct manipulation. The user chooses which agents to make active by dragging and clicking windows, and may minimize or zoom in on agents. Some agents have been delegated a responsibility to perform some task continuously, for example the personal assistant which provides user information after checking agents' credentials. Some agents are delegated one-time tasks, as in the example of the search agent that searches for travel and tourist agents.

This way of combining the two techniques will provide for a powerful system, while ensuring that the user feels active and in control in the interaction process.

The architecture

To support the kind of interaction described above we will need agent servers distributed across the network, much as web servers are today. There will be several types of servers: servers that host service agents, servers that allow visiting agents, and servers for running local and private agents (in-house agents or bots that help the user to interact with other agents). For example, a bank that provides banking services for its customers will provide access agents on their agent server. Instead of accessing account information and transactions using form-based web pages or a Java applet, the bank agent will be used.

Agent servers will provide directory information about the available agents, and the servers will be registered in service directories, provided by server directory agents. To find a suitable agent one may also be able to use the web as a platform for directory information.

Initially, web-based services will be made available as agents by creating agent wrappers around web pages. For this there will be tools that will ensure that the wrappers conform to the common agent format (or communication language).

The research

There are obviously a number of areas of research to pursue to accomplish the Open Agent World. Among them are

- Ontologies of services. Should there be common service ontologies? Maybe the answer is to use dynamic local ontologies in which agents keep their own terminology and agree upon contracts of service after negotiation or after mediation by the user. Maybe directories will classify services according to themes (e.g. the Yahoo web directory) and special mediation agents will translate between the themes.
- Communication languages and protocols. Protocols and languages for administering agents, agent mobility, user agent interaction, etc have to be chosen or designed.
- User-agent interaction. Systems for interaction with agents will be needed (sicsDAIS for example). Interface design issues must be considered. Mobile devices and telephones will be used—how will the interaction with agents take place in these.
- Adaptivity. How will agents learn about their users? Perhaps users may belong to clans that have a common socially built profile. How will personal preferences contrast with the clan's? Will agents provide feedback to the servers after having provided a user with a service, and how will the feedback be used?
- Evolution. Can agents evolve? Can the usefulness of an agent be measured, and if so, can a system for the survival of the fittest be used?
- Security. How can the user's integrity be guaranteed? The ultimate purpose of the agent interaction system is to allow agents to cooperate and share data as effortlessly as possible, which contrasts the requirement of protecting data from unauthorized access. How will agents and servers be protected from each other?
- Trust. How can agents be trusted? There will be a need for trust mechanisms that classify agents and that can use social mechanisms for describing "trustworthiness" (chains of

trust—agent x is trusted by y and y by z and z by a very trustworthy institution). Trust issuing agencies as well as word of mouth directories may be needed.

- Integration of supporting technologies. There will be a need for guarantees of authenticity of security and trust claims using digital signatures etc., systems for secure monetary transactions, etc.
- Legislature concerning the use of agents to represent users. An agent should be able to sign documents, sign contracts, transfer money, etc.

This list is not exhaustive. There are surely more issues to consider. However, the cause seems to be worthwhile in view of the gains that stand to be achieved. Much work remains until it is possible to interact with the infosphere in the manner described in this section, but it is my firm belief that that is where we are headed, in one way or another. Regardless of in what shape the Open Agent World (or whatever the name will come to be), will be incarnated, users will have to interact with the agents. I will be there.

References

1. *Adobe PDF*, Adobe, URL: <http://www.adobe.com/prodindex/acrobat/adobepdf.html>, 1998.
2. André, E., Müller, J., and Rist, T., "WIP/PPP: Automatic Generation of Personalized Multimedia Presenta-tions", In *ACM* Multimedia *96*, pages 407-408. ACM Press, November 1996.
3. Benyon, D., "Adaptive Systems: A Solution to Usability Problems", *User Modeling and User-Adapted Interaction*, nb. 3, pp. 65-87, 1993.
4. Berners-Lee, T. J., Calliau, R. and Groff, J. F., "The World Wide Web", In: *Computer Networks and ISDN Systems*, 24(4-5), pages 454-459, 1992.
5. Bylund, M. and Waern, A., "Adaptation Agents: Providing Uniform Adaptations in Open Service Architectures", *Proceedings of the 3rd ERCIM Workshop on User Interfaces for All*, November 3-4, 1997.
6. Campione, M. and Walrath, K., *The Java Tutorial : Object-Oriented Programming for the Internet*, Second edition, Addison-Wesley Pub Co, Reading, Massachusetts, 1998.
7. Canfield Smith, D., Irby, C., Kimball, R., Verplank, B., Harslem, E., "Designing the Star User Interface", In *Human-Computer Interaction*, Preece, J., Keller, L. (eds), Prentice Hall, Cambridge, England, 1990.
8. Chan, P. and Lee, R., *The Java Class Libraries, Second edition, Volume 2*, Addison-Wesley Pub Co, Reading, Massachusetts, 1997.
9. Charlton, P., Chen, Y., Mamdani, E., Pitt, J., Espinoza, F., Olsson, O., Waern, A., Somers, F., "An Open Agent Architecture for Integrating Multimedia Services", First Autonomous Agents Conference*,* Poster Presentation (Agents 97)*,* Marina del Rey, California, February 5-8, 1997.
10. Charlton, P., Chen, Y., Mamdani, E., Pitt, J., Espinoza, F., Olsson, O., Waern, A., Somers, F. "Open Agent Architecture supporting Multimedia Services on Public Information Kiosks", in *Proceedings of Practical Applications of Intelligent Agents and Multi-Agent Systems*, London, April 1997.
11. Charlton, P., Chen, Y., Mamdani, E., Pitt, J., Espinoza, F., Olsson, O., Waern, A., Somers, F., "Using an Asset Model for Integration of Agents and MultiMedia to Provide An Open Service Architecture", in *Proceedings of ECMAST '97*, Milan, May 21-23, 1997.
12. Cohen, P. R., Cheyer, A. J., Wang, M., and Baeg, S. C., "An open agent architecture", in *AAAI Spring Symposium*, pp. 1-8, March 1994.
13. Dennett, D. C., *The Intentional Stance*. Cambridge, Mass.,MIT Press, 1987.
14. Eriksson, J., Finne, N., Janson, S., "To each and everyone an agent: augmenting web-based commerce with agents", *Proceedings of the International Workshop on Intelligent Agents on the Internet and Web*, World Congress on Expert Systems, Mexico City, March, 1998.
15. Espinoza, F., *A World Wide Web Based Presentation System for an Adaptive Help System*, Masters Thesis, Human-Machine Interaction and Language Engineering Laboratory, Swedish Institute of Computer Science, 1997.

16. Espinoza, F. and Höök, K., "A World Wide Web Interface to an Adaptive Hypermedia System", In: *Proceedings of PAAM '96*, 1996.

17. Feiner, S. K., McKeown, K. R., "Automating the Generation of Coordinated Multimedia Explanations", *Readings in Intelligent User Interfaces*, eds. M.T. Maybury and W. Wahlster, Los Altos, CA: Morgan Kaufmann, 1997.

18. Finin, T. & Wiederhold, G. *An Overview of KQML: A Knowledge Query and Manipulation Language*, Department of Computer Science, Stanford University, 1991.

19. Finin, T., Labrou, Y., & Mayfield, J., "KQML as an agent communication language", In *Software Agents*, ed. J. M. Bradshaw. Menlo Park, Calif.: AAAI Press, 1997.

20. Fritzinger, J. S., Mueller, M., *Java Security*, Sun Microsystems, URL: <http://java.sun.com/security/whitepaper.ps>, 1996.

21. *General Magic | Portico*, General Magic, URL: <http://www.generalmagic.com/portico/portico.html>, 1998.

22. Genesereth, M., Fikes, R., *Knowledge Interchange Format, Version 3.0 Reference Manual*, Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.

23. Genesereth, M. R., and Ketchpel, S.P., "Software Agents", *Communications of the ACM* 37(7): 48-53, 147. 1994.

24. Gosling, J., Joy, B., and Steele, G., *The Java Language Specification*, URL: <http://java.sun.com/docs/books/jls/html/index.html>, 1996.

25. Graf, W. H., "Constraint-Based Graphical Layout of Multimodal Presentations", *Readings in Intelligent User Interfaces*, eds. M.T. Maybury and W. Wahlster, Los Altos, CA: Morgan Kaufmann, 1997.

26. Gruber, T. R., "The Role of Common Ontology in Achieving Shareable, Reusable Knowledge Bases", In: *Proceedings the Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 601-602, 1991.

27. Guttman, R., Moukas, A., and Maes, P., "Agent-mediated Electronic Commerce: A Survey", To appear, *Knowledge Engineering Review*, June 1998.

28. Harrison, C. G., Chess, D. M., and Kerschenbaum, A, *Mobile Agents: Are they a good idea?* IBM Research Report, RC 19887, 1994.

29. Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages", *Artificial Intelligence* 8(3), 323-364, 1977.

30. Horvitz, E., Breese, J., Heckerman, D., Hovel, D., and Rommelse, K., "The Lumiere Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users", *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, July 1998.

31. *HTML 4.0 Specification*, W3C, URL: <http://www.w3.org/TR/REC-html40/>, 1998.

32. *HTTP rfc*, W3C, URL: <http://w3c.org/Protocols/rfc2068/rfc2068>, 1997.

33. *Human Interface Guidelines: The Apple Desktop Interface*, Apple Computer, Inc., Addison-Wesley Publishing, Reading, MA, 1987.

34. Höök, K., *An Approach to a Route Guidance Interface*, Ph.Lic. Thesis No 91-019, Stockholm, Sweden, 1991.

35.Höök, K., *A Glass Box Approach to Adaptive Hypermedia*, Ph.D. Thesis, SICS Dissertation Series 23, ISBN: 91-7153-510-1, Stockholm, Sweden, 1996.

36.Höök K., Rudström Å.,and Waern A., *Edited Adaptive Hypermedia: Combining Human and Machine Intelligence to Achieve Filtered Information*, Presented at the Flexible Hypertext Workshop held in conjunction with The Eighth ACM International Hypertext Conference (Hypertext'97), 1997.

37.Höök, K., "Steps to take before Intelligent User Interfaces become real", In *Interacting with Computers*, In Press. 1998.

38.*JAR Guide*, Sun Microsystems, Inc., URL: <http://java.sun.com/products/jdk/1.2/docs/guide/jar/jarGuide.html>, 1997.

39.*JavaBeans 1.01 specification*, Sun Microsystems, Inc., URL: <http://java.sun.com/beans/docs/spec.html>, 1997.

40.*Java Compiler Compiler (JavaCC) —The Java Parser Generator*, Sun Microsystems, Inc., URL: <http://suntest.sun.com/JavaCC/index.html>, 1998.

41.*Java™ Core Reflection*, Sun Microsystems, Inc., URL: <http://java.sun.com/products/jdk/1.1/docs/guide/reflection/spec/java-reflectionTOC.doc.html>, 1997.

42.Kaehler, T. And Patterson, D. "A Small Taste of Smalltalk", *BYTE*, August, 145-159, 1986.

43.Kay, A. "Computer Software", *Scientific American* 251(3, September), 53-59. 1984.

44.Kay, A. "User Interface: A Personal View", In *The Art of Human-Computer Interface Design*, ed. B. Laurel, 191-208. Reading, Mass.: Addison-Wesley. 1990.

45.Kerr, D., O'Sullivan, D., Evans, R., Richardson, R., Somers, F., "Experiences using Intelligent Agent Technologies as a Unifying Approach to Network and Service Management", In *Proceedings of the 1998 Intelligence in Services + Networks Conference* (IS+N'98) , Springer Verlag, 1998.

46.Kozierok, R., and Maes, P., "A learning interface agent for scheduling meetings", In *Proceedings of the ACM SIGCHI International Workshop on Intelligent User Interfaces*, 81-88. Orlando, Florida: ACM Press, 1993.

47.Lange, D. B. and Mitsuru, O., *Programming Mobile Agents in Java-With the Java Aglet API*, IBM Research, 1997.

48.*Macromedia Director*, Macromedia, URL: <http://www.macromedia.com/software/director/>, 1998.

49.Maes, P., "Agents that Reduce Work and Information Overload", *Communications of the ACM*, Vol. 37, No.7, pp. 31-40, 146, ACM Press, 1994.

50.Maes, P., *User-facing Software Agents*, Tutorial presented at the Practical Applications of Intelligent Agents and Multi-Agent Systems Conference (PAAM'97), London, April 1997.

51.Martin, D. L., Cheyer, A. J., and Moran, D. B.,"Building distributed software systems with the open agent architecture", in *Proc. of the Third International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, (Blackpool, Lancashire, UK), The Practical Application Company Ltd., March 1998.

52.Maybury, M. T. and Wahlster, W. (eds), *Readings in Intelligent User Interfaces*, Morgan Kaufmann Publishers, Inc, San Francisco, California, 1998.

53.*Microsoft PowerPoint*, Microsoft Corporation, URL: <http://www.microsoft.com/office/powerpoint/default.asp>, 1998.

54.Milewski and Lewis. "Delegating to Software Agents", *Int. J. of Human-Computer Studies 46*, 485-500, 1997.

55.*MIME rfc's*, RFC 2045-RFC 2049. Internet Engineering Task Force, 1996.

56.*Mobile Agent Computing A White Paper*, Mitsubishi Electric Information Technology Center America, URL: <http://www.meitca.com/HSL/Projects/Concordia/MobileAgentsWhitePaper.pdf>, 1998.

57.Moran, D. B., Cheyer, A. J., Julia, L. E., Martin, D. L., and Park, S., "Multimodal user interfaces in the Open Agent Architecture", in *Proc. of the 1997 International Conference on Intelligent User Interfaces* (IUI97), (Orlando, Florida), pp. 61-68, 6-9 January 1997.

58.Negroponte, N., "Hospital Corners", In *The Art of Human-Computer Interface Design*. (Ed: Laurel, Brenda) Addison-Wesley, Reading, Massachusetts, 347-353. 1990.

59.Netscape, URL: <http://home.netscape.com>, 1998.

60.*Netscape Plug-in Guide*, Netscape, URL: <http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm>, 1998.

61.Nwana, H. S., "Software Agents: An Overview", *Knowledge Engineering Review*, 11(3): 205-244, 1996.

62.*ObjectSpace Voyager Core Package Technical Overview*, ObjectSpace , Inc., URL: <http://www.objectspace.com/voyager/whitepapers/VoyagerTechOview.pdf> , 1997.

63.*PalmPilot*, 3Com, URL: <http://www.palm.com/home.html>, 1998.

64.Rhodes, B., "Remembrance Agent: A Continuously Running Automated Retrieval System", In: *The Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi Agent Technology*, pp. 487-495, London, UK, April 1996.

65.Shneiderman, B., "Direct manipulation: A step beyond programming languages", *IEEE Comput. 16*, 8 (Aug.), 57-68, 1983.

66.Shneiderman, B., "Direct Manipulation Versus Agents: Paths to Predictable, Controllable, and Comprehensible Interfaces", In *Software Agents*, ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press, 1997.

67.Searle. J. R., *Speech Acts: An Essay in the Philosophy of Language*, Cambridge University Press, 1969.

68.Shoham. Y., "Agent-Oriented Programming", *Artificial Intelligence* 60(1): 51-92, 1993.

69.Shoham. Y., "An Overview of Agent-oriented Programming", In *Software Agents*, ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press, 1997.

70.Smith, S. L., and Mosier, J.N., *Design Guidelines for Designing User Interface Software*, Technical Report MTR-10090 (August), The MITRE Corporation, Bedford, MA 01730, USA, 1986.

71. Van de Velde, W., “Cognitive Architectures—From Knowledge Level to Structural Coupling”, In *The Biology and Technology of Intelligent Autonomous Agents*, ed. L. Steels, 197-221. Berlin: Springer Verlag, 1995.

72. *Virtual Reality Modeling Language*, VRML.ORG, URL: <http://www.vrml.org/Specifications/VRML97/>, 1998.

73. Waern, A., “Service Contract Negotiation - Agent-Based Support for Open Service Environments”, To appear at the 4th Australian workshop on Distributed Artificial Intelligence at AI'98, Brisbane, Australia, 1998.

74. *Windows CE*, Microsoft, URL: <http://www.microsoft.com/windowsce/default.asp>, 1998.

75. Ygge, F., Gustavsson, R., and Akkermans, H., “HOMEBOTS: Intelligent Agents for Decentralized Load Management”, in *Proceeding of DA/DSM '96*, pp. 597 - 610, PennWell Conferences and Exhibitions, 1996.

Appendix A: Script syntax[28]

```
/*
 * White Space
 */

SKIP:
{
  " "
| "\t"
| "\n"
| "\r"
| "\b"
| "\f"
}

/*
 * Comments
 */

SPECIAL_TOKEN: /* COMMENTS */
{
  <SINGLE_LINE_COMMENT: ";" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
}

/*
 * Lexems
 */

TOKEN:
{
  /* Reserved words */
  < OR: "or" >
| < IN: "in" >
| < IF: "if" >
| < AND: "and" >
| < NOT: "not" >
| < COND: "cond" >
| < QUOTE: "quote" >
| < EXISTS: "exists" >
| < FORALL: "forall" | "for-all" >
| < LISTOF: "listof" | "list-of" >
| < SETOF: "setof" | "set-of" >
| < DEFOBJECT: "defobject" | "def-object" >
| < DEFLOGICAL: "deflogical" | "def-logical" >
| < DEFFUNCTION: "deffunction" | "def-function" >
| < DEFRELATION: "defrelation" | "def-relation" >
| < BLOCK: "block" >
| < THE: "the" >
```

[28] This appendix courtesy of Markus Bylund.

```
| < SETOFALL: "setofall" | "set-of-all" >
| < KAPPA: "kappa" >
| < LAMBDA: "lambda" >
| < CONSIS: "consis" >
| < TRUE: "true" >
| < FALSE: "false" >
| < EQ: "=" >
| < NOTEQ: "/=" >
| < IMPLIES: "=>" >
| < IMPLIED: "<=" >
| < EQUIV: "<=>" >
| < RULELR: "=>>" >
| < RULERL: "<<=" >
| < COLEQ: ":=" >
| < COLEQLT: ":=>" >

  /* Basic Lexems */
| < STRING: "\"" ((~["\"","\\","\n","\r"]) |
                      ("\\"(["n","t","b","r","f","\\","'","\""] |
                                  ["0"-"7"](["0"-"7"] )? |
                                  ["0"-"3"] ["0"-"7"] ["0"-"7"]))*
            "\"" >
| < CHARCODE: "#\\" ((["a"-"z"])* | (["0"-"9"])*) >
| < NUMBER: (("-")? <INT> (<EXPONENT>)? ) |
             (("-")? <INT> "." (<EXPONENT>)? ) |
             (("-")? "." <INT> (<EXPONENT>)? ) |
             (("-")? <INT> "." <INT> (<EXPONENT>)? ) >
| < #INT: (["0"-"9"])+ >
| < #EXPONENT: (("E" | "e") ("+" | "-")? (["1"-"9"])+)? >

  /* Variables */
| < INDVAR: "?" <WORD> >
| < SEQVAR: "@" <WORD> >
| < WORD: (~["\n","\r","\t"," ","\"","\\","?","@","(",")","#","'", ";"])+ >
}

/*
 * Root node
 */

SimpleNode CompilationUnit() :
{
  (expression())+ <EOF>
}

/*
 * Expressions
 */

expression() :
{
  form()
| term()
| block()
```

```
}

block() :
{
  "(" <BLOCK> ")"
| "(" <BLOCK> (expression())+ ")"
}

termSeq() :
{
  seqvar()
| term() (term())* [seqvar()]
}

term() :
{
  indvar()
| word()
| string()
| number()
| funTerm()
| listTerm()
| setTerm()
| logTerm()
| quoTerm()
| quanTerm()
}

funTerm() :
{
  "(" word() ")"
| "(" word() termSeq() ")"
}

listTerm() :
{
  "(" <LISTOF> ")"
| "(" <LISTOF> termSeq() ")"
}

setTerm() :
{
  "(" <SETOF> ")"
| "(" <SETOF> termSeq() ")"
}

logTerm() :
{
  "(" <IF> sentence() (term() | block()) [(term() | block())] ")"
  { jjtThis.setOperator(ASTlogTerm.IF); }
| "(" <COND> ("(" sentence() (term() | block()) ")")+ ")"
  { jjtThis.setOperator(ASTlogTerm.COND); }
}
```

```
quoTerm() :
{
  "(" <QUOTE> expression() ")"
| "(" <QUOTE> operator() ")"
| "'" expression()
| "'" operator()
}

quanTerm() :
{
  "(" <SETOFALL> term() sentence() ")"
}

operator() :
{
  termOp()
| sentOp()
| ruleOp()
| defOp()
}

termOp() :
{
  <LISTOF>
| <SETOF>
| <QUOTE>
| <IF>
| <COND>
| <SETOFALL>
| <KAPPA>
| <LAMBDA>
}

sentOp() :
{
  <EQ>
| <NOTEQ>
| <NOT>
| <AND>
| <OR>
| <IMPLIES>
| <IMPLIED>
| <EQUIV>
| <FORALL>
| <EXISTS>
}

ruleOp() :
{
  <RULELR>
| <RULERL>
| <CONSIS>
}
```

```
defOp() :
{
  <DEFOBJECT>
| <DEFFUNCTION>
| <DEFRELATION>
| <COLEQ>
| <COLEQLT>
}

form() :
{
  sentence()
| definition()
}

sentence() :
{
  equation()
| inequality()
| logSent()
| quantSent()
| logConst()
}

equation() :
{
  "(" <EQ> term() term() ")"
}

inequality() :
{
  "(" <NOTEQ> term() term() ")"
}

logSent() :
{
  "(" <NOT> sentence() ")"
| "(" <AND> sentence() (sentence())+ ")"
| "(" <OR> sentence() (sentence())+ ")"
| "(" <IMPLIES> sentence() (sentence())+ ")"
| "(" <IMPLIED> sentence() (sentence())+ ")"
| "(" <EQUIV> sentence() sentence() ")"
}

quantSent() :
{
  "(" <FORALL> indvar() sentence() ")"
| "(" <FORALL> "(" (indvar())+ ")" sentence() ")"
| "(" <EXISTS> indvar() sentence() ")"
| "(" <EXISTS> "(" (indvar())+ ")" sentence() ")"
}

definition() :
{
```

```
  complete()
| partial()
}

complete() :
{
  "(" <DEFOBJECT> word() <COLEQ> (term() | block()) ")"
| "(" <DEFFUNCTION> word() "(" (indvar())* ")" <COLEQ> (term() | block()) ")"
}

partial() :
{
  "(" <DEFOBJECT> word() ")"
| "(" <DEFFUNCTION> word() "(" (indvar())* ")" ")"
}

indvar() :
{
  <INDVAR>
}

seqvar() :
{
  <SEQVAR>
}

logConst() :
{
  <TRUE>
| <FALSE>
}

word() :
{
  <WORD>
}

number() :
{
  <NUMBER>
}

string() :
{
  <STRING>
}
```

Appendix B: Methods of content handlers[29]

Class SE.SICS.HUMLE.PS.ContentHandlerCH[30]

Class hierarchy

```
java.lang.Object
  |
  +----java.awt.Component
        |
        +----java.awt.Container
              |
              +----SE.SICS.HUMLE.PS.ContentHandler
                    |
                    +----SE.SICS.HUMLE.PS.ContentHandlerCH
```

public abstract class ContentHandlerCH

extends ContentHandler

Common methods for Atomic and Composite. This class exists so we can talk about ContentHandler.

Methods

- ContentHandlerCH()

Constructor for a content handler.

- alignment(String)

Sets the alignment of the CH.

- dodbAddedValue(String, String, Vector)

Called when a value has been added to an object in the domain object database.

- dodbAddValue(String, String, Object)

---

[29] This appendix is compiled using JavaDoc from the code documentation of the described classes. All "`---Asset`" methods (`createAsset`, `addAsset`, etc) have changed names to "`---CH`".

[30] The name and the sub-class `ContentHandlerCH` is used to distinguish end-user methods from methods in the super-class `ContentHandler` which are reserved for system use. This could be achieved using access modifiers in one class only, but the two classes are used for documentation purposes; the `ContentHandlerCH` class documentation is available to content handler creators and the `ContentHandler` class documentation is not.

Adds a value to a property in the database.

- dodbChangedValue(String, String, String, Vector, Vector)

Called when a value has changed.

- dodbDefProp(String, String, Object)

Defines a property in the database.

- dodbGetValue(String, String)

Retrieves a value for a property in the database.

- dodbGetValueAtom(String, String)
- dodbRemoveAllEntries(String)

Removes all entries for this ID.

- dodbRemovedProp(String, String, String, Vector)

Called when a property has been removed.

- dodbRemovedValue(String, String, Vector)

Called when a value has been removed.

- dodbRemoveValue(String, String, Object)

Removes a value for a property in the database.

- dodbRemProp(String, String)

Removes a property in the database.

- dodbSetValue(String, String, Object)

Changes a property in the database.

- dodbSubscribe(String, String, Object, String)
- dodbSubscribe(String, String, String)

Subscribes to a value for a property in the database.

- dodbUnsubscribe(String, String)
- dodbUnsubscribe(String, String, Object, String)
- dodbUnsubscribe(String, String, String)

Unsubscribes to a value for a property in the database.

- dodbUnsubscribeAll(Object)

Unsubscribes to all objects for this subscriber.

- dodbUnsubscribeDoObject(String, String)

Unsubscribes to a complete object in the database.

- expand(String)

Controls if the CH should expand and try to be as big as possible.

- finish()

This method does nothing but it can be implemented by You! It is called by the PS system as a part of the process of killing a ch.

- getAllocatedLocation()

Returns the allocatedLocation as a Point.

- getAllocatedSpace()

Returns the actual space of the Component.

- getCurrentCursorType()

Gets the current cursor type.

- getId()

Returns the unique ID of this content handler.

- getMinimumSize()

Returns the minimum size of the CH.

- getPreferredLocation()

Returns the preferredLocation as a Point.

- getPreferredSize()

Returns the preferred size of the CH.

- getPSProperty(String)

Fetches and returns a property from the property file.

- idString()

Gets the id as a string.

- init()

This method does nothing but it can be implemented by You! It is called by the PS system when a ch is created after the constructor has been called.

- invisible()

Sets the visibility of this content handler.

- invisible(String)

Sets the visibility of this ch.

- layout(String)

Sets value of layout for components in this CompositeCH.

- preferredLocation(Integer, Integer)

Sets the preferredLocation.

- sendKQMLMess(String, KifObject, String)

Sends a kqml message to an agent.

- sendKQMLMess(String, String, String)
- sendKQMLMess(String, String, String, String)

- sendScript(String, Object)

Sends a script to be excecuted in another component in the ps.

- setCurrentCursorType(Integer)

Sets the cursor.

- size(Integer, Integer)

Sets value of size.

- trigEvent(String)

Call this in a content handler to fire the specified event.

Class SE.SICS.HUMLE.PS.CompositeCH[31]

Class hierarchy

```
java.lang.Object
  |
  +----java.awt.Component
        |
        +----java.awt.Container
              |
              +----SE.SICS.HUMLE.PS.ContentHandler
                    |
                    +----SE.SICS.HUMLE.PS.ContentHandlerCH
                          |
                          +----SE.SICS.HUMLE.PS.Composite
                                |
                                +----SE.SICS.HUMLE.PS.CompositeCH
```

public class CompositeCH

extends Composite

Methods

- CompositeCH()

Constructor

- addCH(ContentHandlerCH)

Adds a new content handler to this composite.

- createCH()

---

[31] See footnote 30 for a discussion on the choice of names of content handler classes.

Creates a new content handler in this composite.

- emptyCH()

Empty this content handler of components.

- nuke()

This method kills all components of this CompositeCH recursively and resets the CompositeCH.

- padding(Integer)

Sets number of pixels between components.

- removeCH(String)

Removes a specific content handler from this composite.

- replaceCH(String, ContentHandlerCH)

Replaces a content handler.

Class SE.SICS.HUMLE.PS.AtomicCH[32]

Class hierarchy

```
java.lang.Object
  |
  +----java.awt.Component
        |
        +----java.awt.Container
              |
              +----SE.SICS.HUMLE.PS.ContentHandler
                    |
                    +----SE.SICS.HUMLE.PS.ContentHandlerCH
                          |
                          +----SE.SICS.HUMLE.PS.Atomic
                                |
                                +----SE.SICS.HUMLE.PS.AtomicCH
```

public class AtomicCH

extends Atomic This class is the superclass contenthandler of all simple content handlers.

Methods

- AtomicCH()

---

[32] See footnote 30 for a discussion on the choice of names of content handler classes.

Constructor

- nuke()

Nuke kills the content handler.

Chapter 8

Paper B:
sView - Personalized Service Interaction

*s*View – Personalized Service Interaction

Markus Bylund and Fredrik Espinoza

Swedish Institute of Computer Science
{bylund, espinoza}@sics.se
http://www.sics.se/~{bylund, espinoza}

Abstract

The role of the Internet is about to change again. After the hype of web portals we look forward to an Internet in which networked services dominate.

However, the currently prevailing infrastructure for networked services, the World Wide Web (WWW), was not originally developed for handling this type of interaction and many challenges need to be faced. We argue that there are three main problem areas: limited support for service collaboration, producer dominance, and poor support for service interaction in today's web browsers.

A Personal Service Environment (PSE) enables a rich interaction between networked services and their users. In this short introductory text, we present the concept of a PSE in general and the *s*View system (which among other things implements PSEs) in particular.

INTRODUCTION

A PSE, which is private to an individual user, enables user interaction with networked services.

The PSE can store and execute service logic and data, and it can move from computer to computer as the user moves between access nodes within the network.

Preferably, the PSE is run on a computer under the user's immediate control (such as his/her workstation at work or personal computer at home), but if the user is not directly represented on the network, the PSE can run on a server that is dedicated to running PSEs.

The choice of method for interaction between service logic within a PSE and its user is open. The PSE can provide a range of channels for services to interact with their users e.g. HTML over HTTP, WML over WAP, ASCII over SMS, GUIs specified in Swing.

By gathering all services of a user in a PSE, a framework in which services can collaborate is created. This framework lets services share APIs for collaboration, and it can offer support with messaging, ontology handling, authorization etc. The user on the other hand, is given the opportunity to directly control and inspect which services that collaborate and in what way.

Executing the PSE locally makes the user less dependent on a continuous network connection since service providers can upload service logic for local execution. Furthermore, the PSE can provide support for persistence that lets services keep their interaction state. This makes for interaction sessions that can last as the user moves in the network or switches interaction device.

IMPLEMENTATION

The *s*View system is a complete system for user-service interaction that implements PSEs. The system is authored exclusively in Java 1.2, which brings several advantages. The choice of implementation language makes it easier to use the system on different platforms. The popularity of Java makes it easy to reach a broad user group, and finally, Jini technology can be used for naming, searching and distribution of service components.

The *s*View system constitutes a thin layer of infrastructure between the service providers and their users. Nothing is assumed about communication protocols between base services and service components within a PSE, or about interaction protocols between service components and the

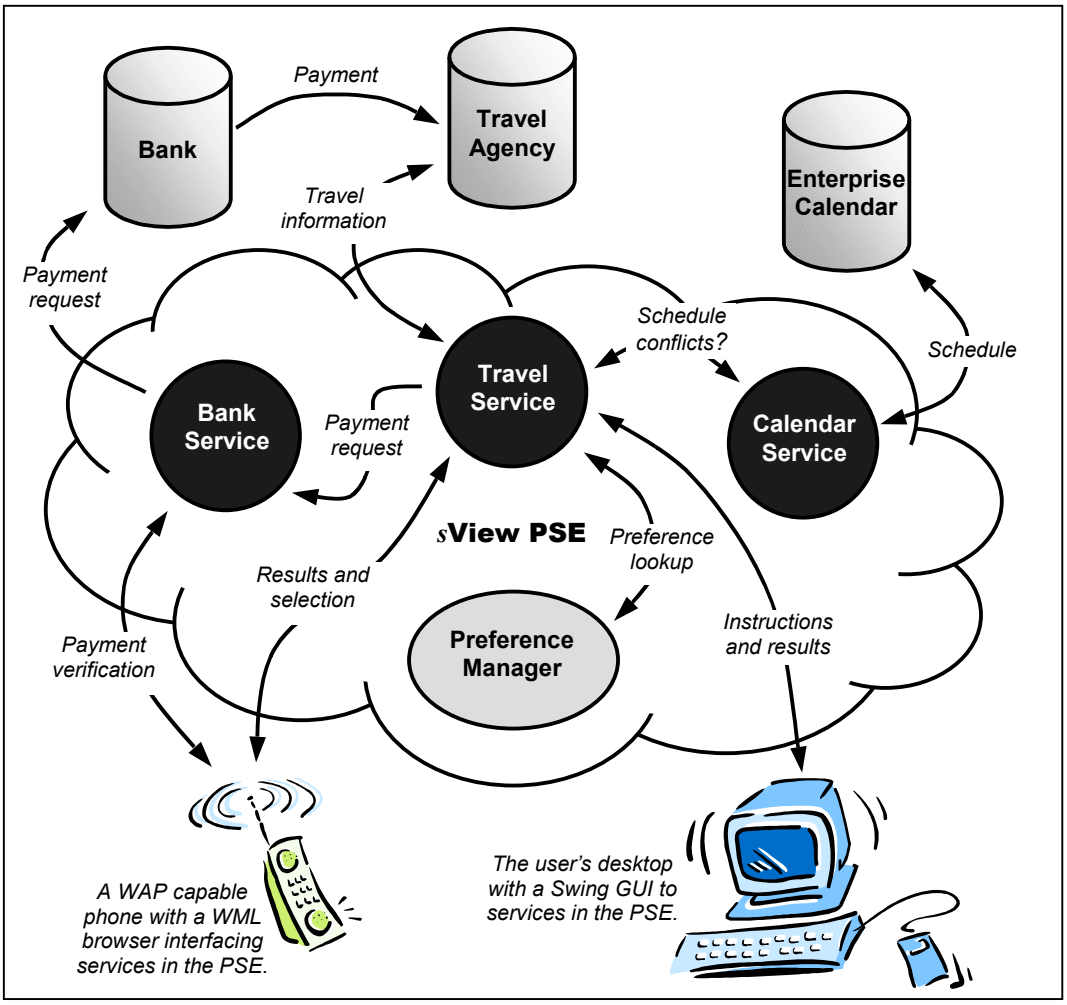

Figure 1. A schematic overview of the scenario described below.

devices that render service presentations. Similarly, no languages, interpreters, or interaction styles are forced upon service components that wish to collaborate within the PSE.

The thin properties of the *s*View system are a deliberate design choice. We want the system to be specific in its overall goal (to enable user-service interaction), but open when it comes to methods of achieving the goal.

SAMPLE SCENARIO

Below we describe a usage scenario[1] that illustrates some features of the PSE (see Figure 1 and 2).

> *A man is about to make a business trip to a foreign city. Using his Personal Service Environment, he locates an electronically mediated travel agency and initiates a dialog with it.*
>
> *The travel agency uploads a travel service component to the user's PSE.*
>
> *Once in the PSE the travel service receives the man's instructions, via a standard graphical user interface (GUI), to make a flight and hotel reservation for his planned trip. Then the man turns his attention to something else and leaves the office bringing his cellular phone.*
>
> *The travel service now makes use of a number of information sources in order to accomplish its task. It searches the PSE for a preference manager and asks it about its client's complete name and address, as well as his seating and smoking preferences. It also*

[1] This particular scenario is operational as a demonstration in the *s*View system.

locates a calendar within the PSE and checks when the man must be back and if the trip conflicts with any of his other appointments.

Having collected all background information, the travel service turns to its base service trying to find an appropriate flight and hotel. The service finds three alternatives that all match the man's request, preferences, and schedule.

The travel agency is now ready to get back to the client with the result of the search. However, since the man is no longer available via the desktop computer (as can be concluded since the screen saver has been on for quite a while), the service contacts him via his cellular phone. The man, now on the train on his way home, selects one of the alternatives and instructs the travel agency to go ahead with the reservation.

The travel service accepts the request and starts searching the client's PSE again, this time for a service that provides payment. One of the man's services, a bank service, is willing to provide payment, but only after a confirmation by the client (this is also done through the interface of the cellular telephone).

Having everything that is needed, including payment, the travel service now executes the man's request by instructing its base service to buy the flight tickets and make the hotel reservation. ■

The above scenario illustrates three important aspects of the PSE.

- The PSE *allows*, and actively *supports*, service collaboration to take place within the environment.
- Since the infrastructure does not *require* HTML/ HTTP, interaction through a number of different devices without loss of interaction state is possible.
- The user shares personal information (such as preferences) with a special purpose component within the PSE (the preference manager). This makes it easy for the user to add, change, inspect, and retract information without having to contact every service that is used. At the same time, services have a central source for such information for every user.

FEATURES

In addition to what the above scenario illustrates, the *s*View system demonstrates the following noteworthy features:

- Service components can be downloaded to the PSE for local execution.
- The use of three different interfaces for interaction with the same service (HTML/HTTP, WML/WAP, GUI/Swing).
- The interaction state can be preserved between usage sessions and throughout the transfer of interaction between devices.
- *s*View allows services to collaborate. In itself it only provides the locale for this collaboration, a locale that is intimately tied to the user. However, it is our intent to do further research in the area of human assisted service collaboration using *s*View as a platform.

CONCLUSIONS

A Personal Service Environment (PSE) enables rich interaction between networked services and their users. We have introduced the concept of a PSE as a solution to many of the challenges with using the WWW for mediation of networked services.

The main contribution of the PSE is that it increases its user's control of networked services. Services are offered support for spontaneous collaboration between peers.

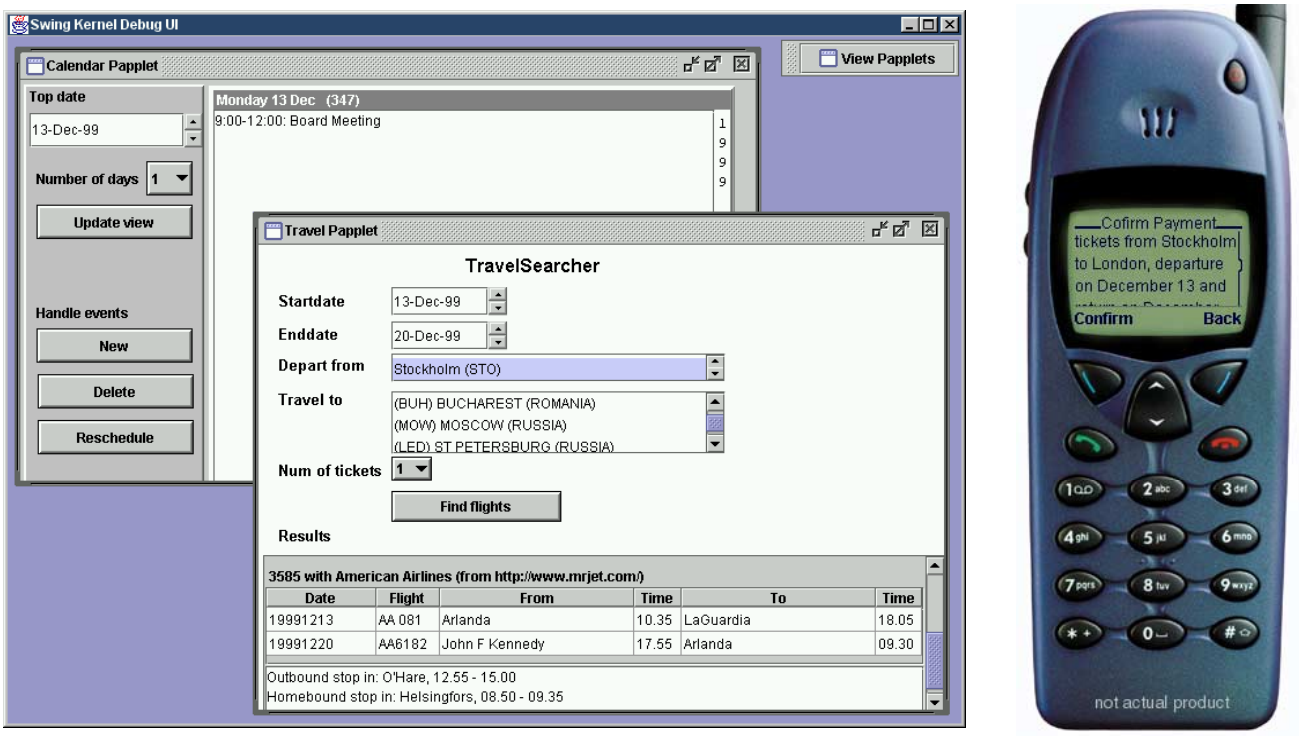

Figure 2. Screenshots from interactions with service components in the sView system. The calendar service and the travel service from the scenario above interact with the user through a SWING GUI. The payment service interacts via a cellular telephone.[2]

The *s*View system will serve as a platform for further research in the area of human assisted service collaboration and personalization of networked services.

ACKNOWLEDGMENTS

The work presented in this paper builds upon the authors' work conducted within the projects Intage[3], EdInfo (Waern, Tierney, Rudström, & Laaksolahti, 1998) and KIMSAC (Bylund, 1999; Charlton et al., 1997; Espinoza, 1998) (funded by the EU ACTS programme) at the Swedish Institute of Computer Science.

REFERENCES

Bylund, M. (1999). *Coordinating Adaptations in Open Service Architectures.* M.Sc. Thesis, Uppsala University, Uppsala.

Charlton, P., Chen, Y., Espinoza, F., Mamdani, A., Olsson, O., Pitt, J., Somers, F., & Waern, A. (1997, April). *An Open Agent Architecture Supporting Multimedia Services on Public Information Kiosks.* Paper presented at the Practical Applications of Intelligent Agents and Multi-Agent Systems, PAAM'97, London, UK.

Espinoza, F. (1998). *sicsDAIS: Managing User Interaction with Multiple Agents.* Ph.Lic. Thesis, The Royal Institute of Technology and Stockholm University, Stockholm.

Waern, A., Tierney, M., Rudström, Å., & Laaksolahti, J. (1998). *ConCall: An information service for researchers based on EdInfo* (T98-04). Kista: Swedish Institute of Computer Science.

---

[2] The snapshot of the cellular phone is taken from the Nokia WAP Toolkit 1.2 that is used to emulate WAP enabled cellular phones (see http://www.forum.nokia.com/developers/wap/wap.html).

[3] See http://www.sics.se/isl/intage/

Chapter 9

Paper C: ServiceDesigner: A Tool to Help End-Users Become Individual Service Providers

© 2003 IEEE. Reprinted, with permission, from Proceedings of HICSS-36, Hawaii International Conference on System Sciences.

ServiceDesigner: a Tool to Help End-Users Become Individual Service Providers

Fredrik Espinoza and Ola Hamfors
Swedish Institute of Computer Science
Box 1263, 164 29 Kista, Sweden
{espinoza, hamfors}@sics.se

Abstract

Pervasive and ubiquitous computing will fundamentally change the mode of interaction between humans and computers; instead of working with applications on a desktop computer we will interact with our environment through electronic services—*anytime and anywhere. In this new modus operandi, specialized and personalized services will become much more important; the usual software house solutions may not be sufficient for individual demands. We propose that end-users themselves can be the service providers; the incentive to create services is grounded in each individual's personal demand for well suited services and this demand will only increase when technology makes it possible to access services ubiquitously.* Individual Service Provisioning *requires three parts: a general platform for managing and accessing electronic services; simple but powerful tools to create the services; and the means to share services between users. Building on previous work developing sView, a personal service environment, this paper presents the second part*—ServiceDesigner—*a tool for creating new services for sView. ServiceDesigner, using web services that expose the functions of web sites as programmatically accessible components, lets end-users create personalized and functional electronic services that fit in the personal platform. With ServiceDesigner, web services are directly available to users and finished services can also be shared with other users.*

1. Introduction

> With the Java model, any and everyone can be a service provider.[1]

We believe computer usage is moving toward a pervasive, mobile, and service centric model. The World Wide Web is evolving from publication of information to provision of interactive services, while mobile communication and computing, including mobile phones and PDAs, are gaining a wider acceptance alongside a wider deployment of interactive services over broadband, cable-TV, and digital-TV networks. In a somewhat more distant future the advancement will move even further, toward a true ubiquitous computing model, wherein traditional computers may disappear completely, to be replaced by intelligent artifacts powered by highly specialized computing devices.

Interactive electronic services and their deployment, delivery, and user access, constitute the

[1] Scott McNealy, founder and CEO of Sun Microsystems Inc., the Java ONE Conference, San Francisco, California, March 2002.

main topics of interest for our ongoing sView project[2] [8, 9]. With the sView system, each user gets a personal service briefcase in which to put their personal electronic services. The briefcase, and the placement of the user's services therein, improves the user's control over services, enables services to adapt to the user and to access-devices, enables services to cooperate, and—with the ServiceDesigner tool—enables users to become individual service providers.

ServiceDesigner is an sView service that lets end-users without special programming skills create new services for sView. In a visual editing environment, the user takes one or more *Web Services* (programmatically accessible and networked functional components), and designs a graphical user interface to the new electronic service. The resulting service is compatible with the sView service environment, and as a self-contained JAR file (Java ARchive—a compressed format for Java files), it may be shared with other users.

Services, in general, may be categorized as being built from scratch, being constructed from a template, being combined from components, or a combination of these.

- Services that a professional developer creates from scratch. This type of service requires significant technical skill on the part of the provider. Much of the work of creating a service is specific to the particular service and the platform for which it is created
- Services constructed from templates. These services are made of pre-constructed service shells that the provider fills with his valuable content to make up the service
- Combinations of building blocks. These are services that the provider has built by combining other services and connecting them to create new functionality
- Combinations of the previous

[2]SView is a project and a system in the OASIS group of the Swedish Institute of Computer Science.

With the ServiceDesigner and the advent of web services, new combinations of services, as in the third type above, will be within reach of end-users; the notion of every user being a service provider is set to become reality. The number of professionally created services for the pervasive and ubiquitous environment will be great but the number of individually created services will be even greater.

1.1. Motivation

The motivation for ServiceDesigner stems from a practical level and a higher more conceptual level.

On a practical level, ServiceDesigner enables users to easily[3] access and directly make use of web services, and to use web services as building blocks when creating new services.

On a high level, we aim for an *Open Service World*[4], in which a user is able to access highly specialized services to fill the needs of any particular moment. Since individual users' needs will differ, either services must have the ability to adapt, or services must be tailored to suit individual users or smaller clusters of users. Large software houses may be unable to adapt and quickly satisfy these needs. This implies an opportunity for small and medium size companies to fill the void, since these companies may be more flexible and better equipped to adapt to users' demands. However, even the more flexible service provisioning of smaller actors may not be enough. If an appropriate service does not exist, just as anyone can publish a web page about themselves or their interests or business, anyone should be able to create a service for themselves or others to use. We call this *Individual Service Provisioning*.

[3]ServiceDesigner is recommended by the Apache SOAP FAQ as a tool to test and directly use web services [4]

[4]The *Open Service World* and *Individual Service Provisioning* concepts will be described in greater detail in a forthcoming (2002) Ph.D. thesis by Fredrik Espinoza. Here we briefly describe these concepts, and the basic underlying architecture, as a backdrop for the ServiceDesigner

To accomplish this we need three different systems:

- The first part is the personal service platform sView [8], which provides a personal service environment for each user. We believe that such as system is neccessary to mitigate the open world of services; without it, the usage of services becomes burdensome, and, concomitantly, the incentive to create services will falter
- The second part, ServiceDesigner, empowers end-users to create their own services
- The third part, a peer-to-peer system called BriefcaseConnectivity [11], provides a generic networking system for services in sView and enables services (including services created by users with ServiceDesigner) to propagate among users in the sView community. More services means more benefit to users and acts as positive feedback to encourage averall usage

The ServiceDesigner is described in the current paper, the other systems are outside this scope and are described elsewhere (e.g. [9, 8, 18] and [11]).

We are also motivated by trying to answer the following questions:

- How do we stimulate the production of a greater range of special purpose and individualized services?
- How can we enable individuals to easily create new service combinations?
- How can we make it possible for unrelated services to come together to provide services above and beyond what they are capable of individually?

1.2. Contribution

ServiceDesigner allows end-users to easily and quickly test web services and to create new services for sView. It allows the user to both create and utilize a *single* web service, or to connect *several* web services to create a new service. A graphical user interface, a web-based interface, and a Wireless Application Protocol (WAP) interface are automatically generated for the resulting service ensuring fast and easy access in many mobile environments.

1.3. Limitations

Please consider the following limitations and caveats when reading this paper:

- It should be evident that there is potential for a great number of services to be available to an sView user. Even if the future platform of choice is not sView, the same types of services will most likely be sought after and therefore produced, albeit for other platforms. Thus, regardless of platform, the reasoning that follows should be applicable. In this text, for reasons of familiarity, we choose to consider sView.
- Surely, *Individual Service Provisioning* also requires an appropriate market model including payment methods that are flexible enough to allow for the types of payments and the amounts that individual service provisioning requires. This, however, is beyond the scope of the current paper.
- There are many ways to describe networked services. We limit ourselves by choosing *Web Services Description Language* (WSDL) [19] because it seems likely that it will become an accepted standard. WSDL is a rich language and it enables you to describe communication of complex objects. For simplicity, we decided not to support complex object communication, limiting the system to communication of simple data types like integers, strings, etc.
- There are many communication protocols for making “remote procedure calls” (RPC). We decided to use *Simple Object Access Protocol* (SOAP) [17], because it uses HTTP as a base transmission protocol and therefore is excellent when communicating over the web.

- Within the Semantic Web research there are efforts similar to the present work [3, 15, 13, 6, 2]. The focus of the web service related work, however, is often on automatic and semantic matching of services to each other. The focus and approach of the present work is on the manual, human assisted matching of services to each other, complemented by sharing of created services. We believe that the two approaches aim for the same goal and that they are complementary.

2. Background

The World Wide Web has evolved from a system for publishing static information in the form of web pages, to providing more and more advanced and interactive services. Unfortunately, the web was not designed for this type of usage, which is evident in, first, the proliferation of add-on protocols and functionality that have been added to the original web infrastructure to keep it running smoothly despite its new requirements, and, second, the advent of web services. Consequently, and in response to the evolution of the web and the growing importance of other domains of electronic services, we designed and developed sView [8, 7, 9]. The following subsections briefly describe sView and web services.

2.1. SView

The idea behind the sView system is to collect all personal services in one place, a personal service briefcase, which is available to the user at all times and from all platforms. The sView platform, with its user centric focus, can alleviate some of the problems of the web as a platform for electronic services and at the same time provide a more flexible and open environment for the future.

The services are kept as close to the user as possible, since local access to services means that we can provide the best possible mode of interaction. Thus, when the briefcase is running on the user's workstation at work or on a PC at home, the user may access services using the graphical user interface. When the user logs out, the services are automatically transferred to a sView Enterprise server in the network, and the user may continue their access using a remote interface such as a mobile phone or a web browser. As long as a user keeps services in their briefcase they are constantly active, even if the user logs out of the system.

There are more benefits that stem from keeping all services in the same environment. First, some services in the briefcase can be used by other services. These generic provider services expand the functionality of the user's briefcase. At the same time they allow service providers to focus on the core business of their services since they can use the shared provider services for functions that are common to many services. Graphical user interface capabilities and peer-to-peer network support has been implemented in this manner.

Next, the user has a greater degree of control over his services when they are all in the same environment. First, the user's personal preferences may be exposed to services as the user chooses and under the user's supervision. All the user preferences can be stored in a preference service to which other services are allowed to connect and retrieve data. Since the preferences are kept in one place, updates are made in one place and then propagated to the necessary services. For example, if the user changes his address he only needs to do it once, in the preference service. Second, with digital signatures and certificates, the user can guarantee the validity and source of a service. Sview has a built in security system with which an un-trusted service can be isolated from the others to preserve the briefcase integrity.

Finally, services can cooperate. This is possibly the most far reaching benefit of sView. The provider service scheme of sharing common functionality is the principal method of service cooperation in sView. During the creation of a ser-

vice, the service provider may implicitly link to provider services that will be required in the briefcase for the service to function properly. However, there is also a way to make services cooperate dynamically. We describe these concepts in greater detail below, in section 3.2 (Connecting Services).

2.2. Web Services

Web services are one type of electronic services. They are modular and programmatically accessible networked components based on XML descriptions and deployed using existing web infrastructure. This new generation of networked electronic services will be available to software developers as distributed pieces of functionality that can be incorporated in new applications as they are being developed. Web services are, however, not being targeted at end-users. Since they inherently have no user interface, only the programmatically accessible interface, they must be incorporated into an application before they can be indirectly used by end-users.

Network calls to web services are often performed using SOAP over HTTP and the services are described using WSDL (although other configurations are possible).

Using SOAP, you literally include a standardized XML-element into a HTTP message where the SOAP specification defines the structure of this XML-element. The XML contains information that enables client/server communication but not as advanced as in the distributed object models. And that is why SOAP is not a replacement for the complex object models - SOAP is a complement, intended to be used when communicating over the web.

A WSDL document describes operations as a set of end-points that can process information. These operations are described in an abstract way, to isolate them from specifications of the kind of communication protocols and formats that are used. The parameters that the operation takes are described with XML-schemas. The document contains all information required to make a callable instance of the service:

- Where the operation is located on the network
- What format is required for the message
- Which protocol should be used when sending the message
- A description of the required parameters
- A description of the response to expect

SOAP and WSDL are key to creating networked services that are programmatically accessible to others.

3. ServiceDesigner

We now take a closer look at the ServiceDesigner system.

The ServiceDesigner should assist the user in the following steps:

- The first step is to let a user access web based service descriptions (WSDL documents). The user should be able to get information on what kind of functionality the service provides (i.e. its *functional components*) and then be able to choose which functionality to use. It should be possible to choose functional components from several different web services
- The next step is to automatically generate a default graphical user interface for the chosen functionality. The user should be able to modify and design the graphical user interface with a visual tool. The service should be as accessible as possible, i.e. have several different graphical interfaces like HTML, WML, JAVA-SWING etc. As soon as the user interface has been generated it should be possible to test the service with real parameters
- In the last step the designed user interface together with a specification of the functional

components that have been used is combined into an sView compatible service. This step involves integrating code that is necessary to display the interface, make the SOAP function calls, and to function as an sView service. The integration results in an Sview service that is ready to be loaded into the personal service briefcase or distributed to other users

- We also complement the basic functionality described above with support for connecting together several functional components in one coherent service. The user should have some (preferably total) control over the information flow between the parts of functionality. We describe this further below, in section 3.2.

Another design goal was to make the ServiceDesigner easy to use. This means that an end-user should be able to use it without writing any code. The ServiceDesigner is designed with these criteria in mind.

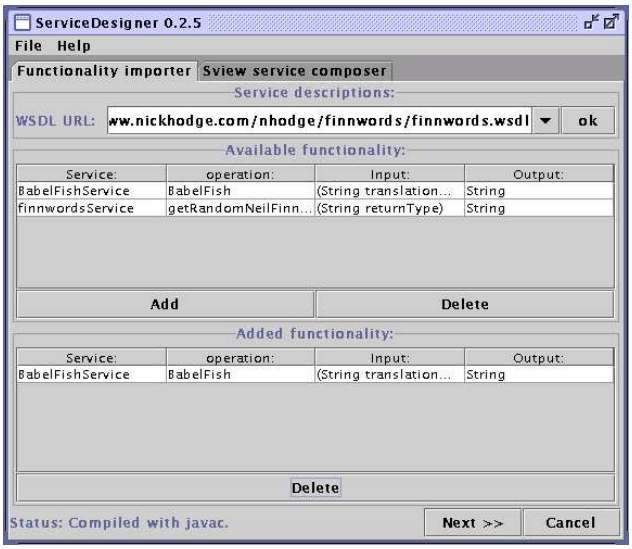

Figure 1. The main interface window of the ServiceDesigner. In this interface the user inputs a URL to a service description document which is loaded and understood by the system. The user then chooses from a list of available functional components (items in the middle area)

3.1. Creating a Basic Service

The first step in creating a new service is to load the service descriptions of the component parts (see Fig. 1). The URL of the WSDL document is entered into the top text field of the main window of ServiceDesigner. When the "Ok" button is clicked, the WSDL document is loaded and a list of available functional components is displayed in the "Available Functionality" area. The user can choose from this list and add functional components to the "Added Functionality" area below. The chosen functional components will be included in the service and its interface when the user clicks the "Next" button to go to the next step, generating the interface.

3.1.1 Generating the Interface

When the interface is generated each functional component is represented by a set of text fields and labels, and an activation button. In Fig. 2 we see that one functional component was chosen.

The text fields with the corresponding labels represent input parameters to the functional component. The labels are extracted from the WSDL document and tend to be more or less descriptive. The activation button is generated in such as way that clicking it activates the function with the contents of the text fields as parameters. There is also a generic output area into which is put any potential output that is the result of a function call. Testing the service entails filling in the text fields and clicking the activation button. A SOAP call will then be carried out and the result of the call will be presented in the output area.

The default generated graphical user interface can be modified by the user, which means that the user can change the default graphical components to something more appropriate. A text field may

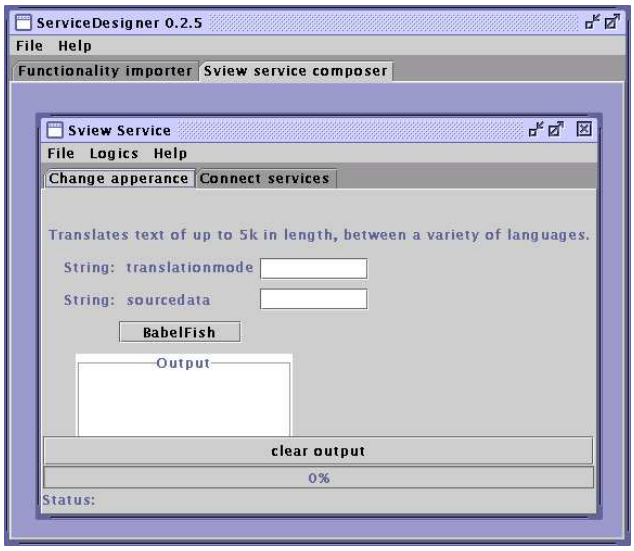

Figure 2. The generated interface for one functional component.

be changed to a text area for more space or to a selection box (pop-up menu) for pre-defined values. The user can also set the value of a text field to be final, which implies that the same value will always be used.

Moreover, label texts can be edited, positions and sizes of components can be changed, and the output area size and position can be modified.

3.1.2 Generating the sView Service

Finally, when the user is happy with the design of the interface, the sView service can be generated. The user choses File/Generate Service and then fills in a dialog asking for a name of the service, the authors name, keywords, and any other text that can help a potential future user to understand and make use of the new service.

We chose to generate the service as JAVA-source code that can be compiled with generated scripts. The scripts also generate a JAR file version of the service. This JAR file, which can be distributed to other users, contains everything that is needed in order to run the service in sView.

After the service has been generated it is ready to be used and it is automatically loaded into sView.

3.2. Connecting Services

One of the most significant properties of the ServiceDesigner is its ability to connect together several different functional components from several different web services. This means that completely unrelated services can be combined into never before seen combinations simply according to the needs and taste of the user. When such a combined service is created it too may be shared with others, and just as in the case of a single component service, other users of the service do not need to concern themselves with the underlying functionality or couplings between disparate web services.

To create a combination of web services we first we need a graphical representation of the functional components and we decided on gray boxes containing the service name (see Fig. 3. This representation resembles the way the Hive system [14] represents its agents.

To connect two functional components X and Y the user holds down “Shift” and drags the mouse from one box to another. This implies that the output of the first functional object should become one of the inputs of the other. But the second component may have several input fields which prompts the connection dialog to be displayed (Fig. 4).

The connection dialog asks the user if he wants to connect the output of X to one of the parameters of Y. The user then selects one parameter that makes sense and the connection is complete. The connection is represented as a graphical arrow (see Fig. 3).

It is the logic capability of the user that dictates if the connection will work or not—the system will not stop a user from making illogical connections. However, it is a simple matter to test connections to make sure they work properly, and

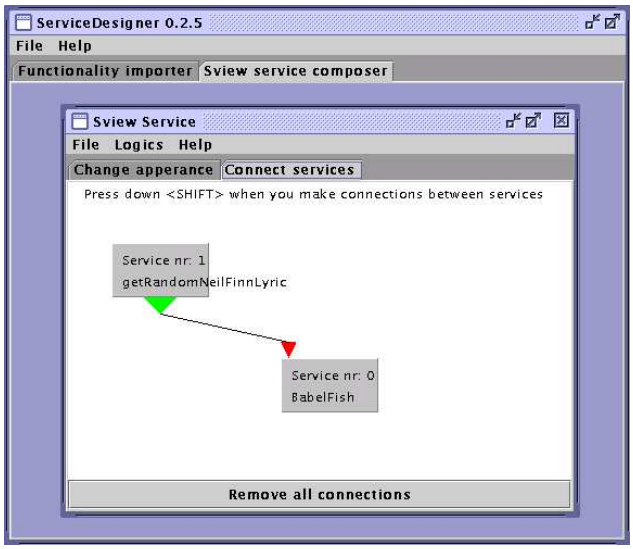

Figure 3. Connecting two different functional components.

Figure 4. When connecting the output of one functional component to another the user must choose to which input the connection should go.

consequently, in particularly difficult cases, trial and error can always be used.

When a connection is made it affects the graphical components (that were created to interact with the service) accordingly:

Assume that X, Y and Z are functional components that require parameters (X^1, X^2, ... X^n), (Y^1, Y^2, ... Y^n) and (Z^1, Z^2, ... Z^n).

- If X is connected to the parameter Y^1 of Y, the graphical component that represents Y^1 will disappear because it becomes superfluous (the user does not have to write in the value. It is taken from the output of X)
- If X is connected to Y, the button that invokes X will disappear. This happens because the button of Y will invoke X when it is pressed (chain reaction)

We also need some logic to handle situations where connections are not allowed. This logic should function as follows:

- If there is a connection between X and Y, i.e. the X output is connected to a parameter of Y, the user should not be able to make a connection the other way around i.e. from Y to X (direct feedback), because this would lead to an infinite loop
- If X is connected to Y and Y is connected to Z neither Z nor Y can be connected to X (indirect feedback in the Z case and direct feedback in the Y case). This would lead to an infinite loop
- Only output from one functional component can be connected to a parameter. When a connection is made the parameter is considered occupied

Trees of connected functional components are executed recursively starting from the functional component that still has an activation button (see Fig. 5).

With a tree like that in Fig. 5 the following execution order would be possible:

1, 4, 2, 3, 5, 6

Or, depending on the order in which the connection was made, i.e. if 5 was connected to 6 before 4 was connected to 6:

2, 3, 5, 1, 4, 6

If you create a connection between 2 and 4 the execution order changes like this (Fig. 6):

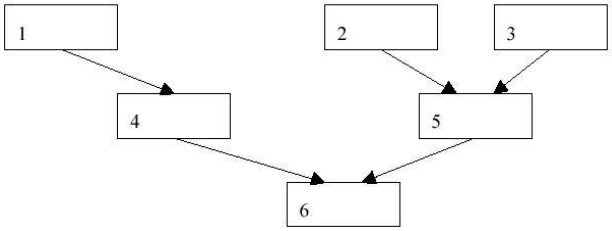

Figure 5. The connected functional components. With a tree like this functional component 6 has an activation button.

1, 2, 4, 2, 3, 5, 6

Or

2, 3, 5, 1, 2, 4, 6

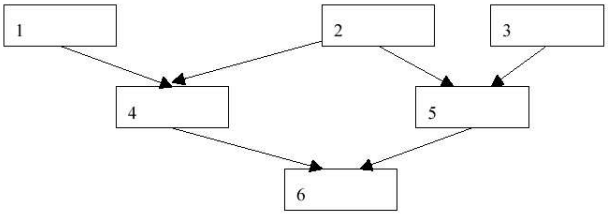

Figure 6. A new connection has been created between 2 and 4.

Notice that functional component 2 has been executed twice in both sequences. This means that more executions than necessary have been made, e.g. every time a "tree" executes every functional component only needs to execute once.

Introducing a session-state, which holds information about the functional component (if it has been executed or not), solved this problem. The execution order changed to be the same as when we did not have a connection between in 2 and 4 (see Fig. 5).

3.2.1 Built-In Functions

When you create services you sometimes need to add very simple functions such as basic arithmetic or logic. In those cases it is better to use built-in functions. For example, it is not necessarily efficient to make a network call to to add two numbers, it can be done on the client side.

Here are some of the built-in functions we have implemented:

- A repeater. With a repeater you can schedule repeated invocations of a web service or a execution tree. For example, it is possible to schedule stock quotes to be sent by Short Message Service (SMS) every hour
- Arithmetic. We include the four basic arithmetic operators

4. Sharing Services

Given the creative freedom that the sView architecture provides, it is conceivable that the body of future sView users will generate a large number of different services. With the ServiceDesigner it is easy to share these services among users: the creator of a service can publish the service to a web site for others to download, or he can simply send the service to another user by electronic mail or other electronic means. In the Future Work section below we will touch upon another such possibility.

However, the ServiceDesigner has a built-in feature that directly enables a user to become an individual service provider. Each service that is generated using ServiceDesigner is given its own SOAP enabled interface. After the service has been generated and it is loaded into sView, it can expose a new SOAP interface that corresponds to whatever parameters and function the newly created service exhibits. This feature enables the sView briefcase to act as a web service end-point exactly in the same manner as other web service engines.

5. Related Work

The concept of combining pieces of information or functional components into a new composite service may be found in several other systems of which we will mention two: *Data Tiles* and *InfoBeans*.

5.1. Data Tiles

The DataTiles system [16] is a platform where end-users place plastic tiles on a big touch-sensitive screen. As soon as a tile is laid on the screen it becomes interactive and it is possible to lay several tiles adjacent to each other to form combinations.

DataTiles is a modular system in which the designers have tried to integrate physical and graphical interaction. To accomplish this, the designers divided the system into three parts:

- Transparent plastic tiles that each has a unique ID
- A large touch sensitive screen, which identifies the plastic tiles when they are laid on the screen
- A computer connected to the system

When a tile is placed on the screen it is identified and a program (i.e. service) that has previously been associated with the tile starts in the computer. The program creates a graphical interface exactly under the tile and the area of the tile is lit up. Users can then interact with the service by touching the tile with a special pen.

The idea behind DataTiles is that the tiles should function as small entities, which can be used separately or be combined into more complex applications—you simply put the tiles next to each other and they automatically recognize each other and start to communicate.

The user is encouraged to create logical chains, just like you create complex sentences by using small words.

DataTiles is similar to ServiceDesigner in that end-users are able to combine tiles (or functional components as we call it) into new types of services with a common graphical interface.

However, the difference is that we dynamically build the graphical user interface for each functional component (which corresponds to a tile), which means that we can use services that were not intended (or known ahead of time) for the ServiceDesigner. A DataTile (used like a service) has to be tailor-made for the system—if you were to place a tile with an unknown ID on the screen, the system would not know what to do. And, in addition, all required combinations of tiles must be pre-defined. Since each tile (or rather the service represented by the tile) has a known interface that describes its functionality, to connect to it, another tile must use this interface.

On the other hand, ServiceDesigner services are less interactive than their DataTile counterparts. Our services are more akin to "pressing the submit button" as this is the only possible event. This does, however, result in a system that is more open and flexible to new functionality, and in services that are appropriately interactive for a mobile or ubiquitous setting.

5.2. InfoBeans

InfoBeans [5] lets end-users configure their own individualized information services from different web sites.

An InfoBean is a container that holds a small part of an existing web page. The user selects which part of the page the InfoBean should hold. The system then trains itself with the help of the user to understand how to parse out the right information even though the page has changed i.e. to learn the structure of the web page and therefore be able to understand how to handle the category of web pages that it represents.

By collecting several infoBeans in an infoBox (a DHTML based web page) the information from several web pages can be gathered. Every infoBean has input and output channels, which makes it possible to connect infoBeans to trans-

fer data from one infoBean to another.

The infoBean system is conceptually similar to ServiceDesigner together with Sview in that both systems use independent services that can be combined in one common graphical user interface. But there are a few things that differ.

The most fundamental difference is the choice of functionality. InfoBeans uses heuristic methods in order to find the wanted content in the web page; ServiceDesigner uses strictly defined and well described web-based functionality[5], which leads to a more reliable system. We do not have a "server-side" that needs to process heuristic methods leading to scalability problems if you cannot distribute the processes on the "client-side".

Second, we dynamically build a graphical user interface to the parameters of the service—the InfoBeans system renders HTML in small windows, which leads to a system that bears resemblance to many small and simultaneously open browser windows. We have total control over the actual components and can therefore change and arrange them freely.

Another difference is that the InfoBeans system lacks an overlying framework like sView. When using sView we can have other services, not just simple information services, open at the same time.

6. Future Work

Thanks to the current capabilities of ServiceDesigner it will hopefully be easy for end-users to create and exchange services with each other. Furthermore, it is obvious, based on the success of today's popular peer-to-peer file sharing applications, that sView users would benefit from a network in which they could more conveniently share their services with each other. We envision using the built in peer-to-peer support (BriefcaseConnectivity, as outlined in section 1.1) of sView to bring complete service availability to the sView platform.

There are two ostensible advantages in creating a network of sView briefcases:

- Users will have continuous and ubiquitous access to new information and services
- It will be possible to design services that leverage the community of sView briefcases including, but not limited to:
  - Trust chains built on the linking of trust from user to user within the sView community (c.f. [1])
  - Rating services that track the performance and quality of services
  - Social mechanisms for recommending services
  - Alert mechanisms that build on the experiences of members of the community to protect the community as a whole from malevolent services

A phenomenon common to many peer-to-peer networks is described by what economists call the network effect [10, 12]. The network effect states that as the network increases in size, the value of a network to an individual also increases. That is, as more and more resources enter the network, the more valuable the network is to the individual. We hope to generate our own network effect by enabling sView users to participate in a network of sView briefcases, thereby stimulating service creation and use. The purpose of this work is to create a network comprised of sView briefcases that supports the sharing of services and which protects individual users by a number of trust mechanisms.

When the number of services increases, there is a risk that it will be harder to ensure their quality and the user's security. And with increasing numbers of individually created services, we can no longer depend on the good standing and reputation of the service provider when deciding to use or not use specific services. We therefore believe it is of great importance to provide

[5] The format or syntax, but not necessarily the semantics, are well-defined

a platform such as sView/ServiceDesigner with tools and support for trust of services. In a separate project which is scheduled to commence late 2002 we will design and implement such support.

7. Summary and Conclusions

We wish to leverage the use of web services to empower users to become individual service providers, and to demonstrate our ideas we have developed an sView service called ServiceDesigner. It assists the user in the following steps: First, it lets a user access web service descriptions. This entails getting information on what kind of functionality the service provides and then being able to choose which functionality to use. Second, it automatically generates a default graphical user interface for the chosen functionality. The user can visually modify the graphical user interface to his liking. Finally, ServiceDesigner takes the web service description and the designed user interface and generates the necessary programming code to make the new service sView compatible. When the integration is finished we have an sView service that can be loaded into the personal service briefcase or distributed among other users. The finished service also exports its own service description, as a web service, thus making the user's sView briefcase act as a web service provider.

By enabling users to produce sView-compatible services with ServiceDesigner, we increase end-users' freedom of combining, individualizing, and personalizing web service based functionality. We believe that the web service architecture is the web-developer model of the future, where professional service providers develop small pieces of functionality that end-users, with the assistance of different tools—like ServiceDesigner—can combine into personalized services. More services implies more benefit to users and with the positive feedback of a growing service base we may well see end-users becoming individual service providers.

8. Acknowledgments

This work was partially financed through the sView SITI[6] project [18]. The authors wish to thank the rest of the OASIS team at SICS, including Markus Bylund, Anna Sandin, Lucas Hinz, Stina Nylander, and Andreas Espinoza.

References

[1] K. Aberer and Z. Despotovic. Managing Trust in a Peer-2-Peer Information System. In H. Paques, L. Liu, and D. Grossman, editors, *Proceedings of the Tenth International Conferenceon Information andKnowledgeManagement(CIKM01)*, pages 310–317, New York, Nov. 2001. ACM Press.

[2] A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. Sycara. DAML-S: Web Service Description for the Semantic Web. In I. H. . J. Hendler, editor, *The Semantic Web - ISWC 2002*, Lecture Notes in Computer Science 2342, pages 348–363. Springer Verlag, 2002.

[3] A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, S. A. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. DAML-S: Semantic Markup for Web Services. In *International Semantic Web Working Symposium (SWWS)*, July 2001.

[4] Apache SOAP FAQ. Web, 2002.

[5] M. Bauer and D. Dengler. InfoBeans - Configuration of Personalized Information Services. In *Proceedings of the International Conference on Intelligent User Interfaces*, pages 153–156, 1999.

[6] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[7] M. Bylund. Personal Service Environments - Openness and User Control in User-Service Interaction. Licentiate of Philosophy Thesis, Uppsala University, 2001.

[6] Swedish Information Technology Institute

[8] M. Bylund and F. Espinoza. sView – Personalized Service Interaction. In J. Bradshaw and G. Arnold, editors, *Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 2000)*, pages 215–218, Manchester, UK, Apr. 2000. The Practical Application Company Ltd.

[9] M. Bylund and A. Wærn. Personal Service Environments - Openness and User Control in User-Service Interaction. Technical Report T2001:07, Swedish Institute of Computer Science, Kista, Sweden, May 2001.

[10] J. Farrell and G. Saloner. Standardization, Compatibility, and Innovation. *RAND Journal of Economics*, 16:70–83, 1985.

[11] L. Hinz. Peer-to-Peer Support in a Personal Service Environment. Master's thesis, Uppsala University, Uppsala, Sweden, 2002.

[12] M. Katz and C. Shapiro. Network Externalities, Competition, and Compatibilty. *American Economic Review*, 75:424–440, 1985.

[13] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[14] N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes. Hive: Distributed Agents for Networking Things. In *Proceedings of ASA/MA'99, the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, 1999.

[15] S. Narayanan and S. A. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proceedings of the Eleventh International Conference on World Wide Web*, pages 77–88. ACM Press, 2002.

[16] J. Rekimoto, B. Ullmer, and H. Oba. DataTiles: A Modular Platform for Mixed Physical and Graphical Interactions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 269–276, New York, NY, USA, 2001. ACM Press.

[17] SOAP 1.1 Specification. Web, 2001.

[18] SView. Web, 2002.

[19] WSDL 1.1 Specification. Web, 2001.

Chapter 10

Paper D:
Generic Peer-to-Peer Support for a Personal Service Platform

© 2003 IEEE. Reprinted, with permission, from Proceedings of the 2003 International Symposium on Applications and the Internet (SAINT 2003).

Generic Peer-to-Peer Support for a Personal Service Platform

Fredrik Espinoza Lucas Hinz

Swedish Institute of Computer Science
Box 1263, 164 29 Kista, Sweden
Telephone: +46 8 633 1500
{espinoza, lphinz}@sics.se

Abstract

Building on previous work of the OASIS group at SICS, this paper presents a generic peer-to-peer system for the sView personal service platform. The sView platform provides each user with a personal service briefcase from which services may be reached using a variety of devices and interfaces. With the peer-to-peer system, called Briefcase Connectivity, we leverage the community of all sView users, the ultimate purpose of which is to simplify the propagation of cooperating services, to enable users to become individual service providers, and to bring to the users a greater number of specialized services. We suggest that sView with Briefcase Connectivity and today's electronic services demonstrate a viable model to make a step toward tomorrow's ubiquitous computing.

1 Introduction

On a daily basis people use a wide range of *electronic services* to enhance their lives—to communicate with friends and colleagues, to make travel reservations, or to access information. Electronic services may be found in many different networks: on the World Wide Web, in telecom operators' networks (e.g. voice mail), in home area networks (e.g. alarm control and home entertainment systems), or in interactive digital cable and TV networks. Unfortunately, the design of electronic services seldom focuses on the user, but instead focuses on the service provider or the devices used to access the services [5]. Recognizing this oversight, the OASIS group at SICS[1] developed the concept of a Personal Service Environment, or PSE [3]. The PSE is intended to make the use of electronic services easier for the user, and for this reason focuses on increased user control, enhanced service interoperability, and support for continuous, ubiquitous access to the user's whole set of services on a wide variety of devices. Conceptually, a PSE is an individually collected and tailored set of services, available to the user at all times, and at least partially independent of Internet access. The reference implementation of the PSE, called sView [4], is a virtual *briefcase* that serves as a storage and execution environment for electronic services.

SView is equipped with a number of mechanisms that mitigate the design of new services and facilitate service interoperability. For example, the *ServiceDesigner* service [6, 8] enables the user to collect and combine Web Services[2] into a single, collaborative service unit, furnish it with a

[1] Swedish Institute of Computer Science, http://www.sics.se.

[2] http://www-106.ibm.com/developerworks/webservices/.

graphical user interface, and generate a fully qualified sView service for use in the user's briefcase. This gives the user the freedom to fill his briefcase with services catered to his needs.

1.1 Motivation

Given the creative freedom that the sView architecture provides, it is conceivable that the body of sView developers and users in a near future will generate a large number of different services. Based on the success of today's popular peer-to-peer file sharing applications, it is obvious that sView users would benefit from a network in which they could share their services with each other. In one of our scenarios entitled *Individual Service Provisioning* each user can be a service provider. Using a tool like the ServiceDesigner the user creates a service and then shares it with other users in the sView community. This is the primary motivation for the present work.

But peer-to-peer computing is advantageous in other areas besides service sharing, exemplified by the multitude of systems that utilize peer-to-peer[3]. Ranging from instant messaging applications to wide-area storage architectures, new systems are being developed that incorporate peer-to-peer in their design. These systems leverage peer-to-peer to enhance system performance, increase service availability, and to guarantee user anonymity.

To cater to both of these situations, we wish to extend the functionality of the existing sView platform with a generic peer-to-peer based communications layer. This capability, in the form of a service called *Briefcase Connectivity*, will bring sView users together in an always-on network of sView briefcases.

With Briefcase Connectivity it will be possible to design services that leverage the community of sView briefcases including, but not limited to: information and resource sharing services; collaborative services such as bulletin boards, chat rooms, and auctions, and other services that include social mechanisms, such as trust chains, real-time ratings, etc.; and distributed services that tackle complex problems.

[3]In this paper, the terms peer-to-peer, peer-to-peer computing, and P2P denote the same concept.

1.2 Background

Two developmental trends characterize what is happening to the Internet today. First, the Internet is becoming increasingly service-oriented. Nearly every public and private institution has a web site today, and most of these give the user access to some type of service—to manage bank accounts, to book hotel reservations, or to search a directory for an address. Another indication that the Internet is becoming service-oriented is the Web Services movement, led by Ariba, IBM, Microsoft, and others. Web Services are intended to make web sites more modular, allowing service providers to make their web sites programmatically accessible. That is, service providers will have access to service functionality via the Web, facilitating the design of new services.

With the increased focus on *service oriented computing*, we argue that an environment like sView will be beneficial to users, as a unifying environment for all services. Thus, we here introduce the main concepts of PSEs and sView for the purpose of giving a backdrop to the present work on Briefcase Connectivity. For more details regarding lessons learned, the novelty of sView relative other service frameworks, and a more precise description of the technical details of sView, please see [3, 4, 5].

In sView, users can store, execute, and even create electronic services in a personal, virtual briefcase. Services can be reached using a variety of devices and platforms (including graphical user interfaces on ordinary PCs, web-based interfaces, and interfaces in mobile phones), and the accompanying sView server architecture ensures that a user's briefcase is always continuously running and ready at hand. SView is attractive be-

cause it collects all of a user's services in a single, uniform environment, freeing the user from having to alternate between service environments to access services.

The unifying theme for the sView system is user control, defined by openness, continuous and ubiquitous access, personalization, and collaboration.

Users should have complete freedom regarding which services they use in their briefcase. For this reason, sView's service architecture is open, meaning that any service provider, including the user, can design services for sView.

Services should be available to the user at all times and should be accessible on a variety of devices. The sView briefcase is accessible on a wide range of devices including laptop computers, PDAs, and mobile phones. SView achieves continuous access by enabling services to save their state, making it possible for the service briefcase to migrate between usage sessions and devices without having to restart the services it contains.

Many electronic services require and collect personal information about the individual user. The sView framework makes it easy to monitor and control which services have access to what information.

One of the shortcomings of many (proprietary) services is their inability to interoperate. The sView architecture promotes service interoperation by making it possible for services to share APIs with each other. Another level of interoperation is achieved by the ServiceDesigner, which allows a user to connect Web Services to each other and to other services in sView.

To further illustrate sView, let us consider a usage scenario: our example user Michael is working in his office and keeps his sView briefcase open and running on his office PC. In the briefcase he has a number of services including email, instant messaging, booking service for the spa in his apartment building, pizza delivery, airline booking, and so on. At five o'clock, Michael decides to leave work, and logs out of the briefcase and walks to the subway. As he logs out, the services in the briefcase are stopped, compressed, and transferred to a central server for offline access—arriving at the server, the services are once again started and made active. While on the train, Michael decides that he feels like a soak in the spa and he therefore takes out his cell phone and gains access to his briefcase remotely, using the phone's wireless Internet connection. He checks the bookings and after finding the spa completely available for the whole evening, books the next two hours. He also decides to order a pizza to go with his soak in the tub. When he gets home, he gets the idea to invite his girl friend for the evening. Using his tablet PC, he logs into sView and first fires off an instant message with the proposal; his girlfriend, also being an sView user, immediately responds with a yes. Then he checks on the pizza order; the pizza has not yet been dispatched so he changes the order to a larger size. Finally, he changes the spa booking, adding two more hours. When Michael logs into sView on the tablet PC, it connects via the wireless home network to the remote server and fetches the active briefcase. Once again, Michael can interact with his services using more powerful graphical user interfaces, since the services have been brought to the local machine where they are now executing; thanks to the sView server architecture, the session has never been interrupted and the services have kept their state.

The second developmental trend of today's Internet is marked by peer-to-peer's emergence as a competitive computing model. P2P's utility has historically been overshadowed by application designers' preference for the client-server model. The recent success of large-scale file sharing applications has rekindled an interest in peer-to-peer, compelling commercial and private industry alike to develop software that incorporate peer-to-peer in their design.

1.3 Paper Overview

In the next section we describe Briefcase Connectivity. Following this, we examine a use case involving a service that utilizes Briefcase Connectivity—Sentinel. The next section describes related work, and finally, in Conclusions, we summarize our work and give some hints as to possible future work.

2 Briefcase Connectivity

Briefcase Connectivity is a JXTA[4]-based generic peer-to-peer communications system for sView (more on JXRA below). It is intended to solve two problems. The first problem involves building and maintaining a network of sView briefcases. The second problem is providing a general communication mechanism that sView services can use to communicate over the sView network. On the one hand, Briefcase Connectivity must provide a component that is accessible to other briefcases via the network. On the other hand, it must provide an interface accessible by sView services contained in the briefcase.

Before we examine the design and implementation of Briefcase Connectivity let us consider the following questions:

What is the novelty of Briefcase Connectivity? Briefcase Connectivity is essentially a generic peer-to-peer system for the ordinary user. In our design and construction of the system we have aimed at making it as simple as possible to use. At the moment it is easy for a developer to use Briefcase Connectivity, and in our "Individual Service Provisioning" scenario[5], where every user can be a service provider, some of the services produced can easily be peer-to-peer enabled thanks to Briefcase Connectivity. By creating a network of sView users, who all contribute with services and resources, we hope to generate a network effect with positive feedback [12] where more services contributed by members leads to higher incentive to participate in the community.

What can you do with Briefcase Connectivity in sView? With support for peer-to-peer, service providers can leverage the potential small world effects of communities [1] and produce services that add value to sView users. As we will see below, with Briefcase Connectivity it is relatively easy to create a peer-to-peer enabled service or to add auxiliary peer-to-peer functionality to an existing service. Consider a standard MP3 player such as Winamp[6]. Now imagine adding to Winamp a peer-to-peer system in order to present to the user a top list of the songs being played, in all instances of Winamp, right now—or in other words the "Real-Time Top 10"[7]. To do this you would need to build a complete peer-to-peer system including the protocol design, network interfaces, discovery, caching, addressing, etc.—a lot of work to add a little function. In sView, with Briefcase Connectivity, you simply call the send method of the Briefcase Connectivity API. This broadcasts information about which song you are currently playing to other instances of the MP3 player. Each player in each briefcase then builds its own top list by compiling the information being sent within the network of MP3 players.

How does Briefcase Connectivity differ from Jini/RMI/Sockets/...? Briefcase Connectivity is at a higher level of abstraction. This makes it easier for developers to create services that make use of the peer-to-peer network. Furthermore, Briefcase Connectivity directly gives the developer and end-user access to a peer-to-peer system

[4] Project JXTA: http://www.jxta.org.

[5] Individual Service Provisioning is described briefly in Sect. 1.1. A more thorough discussion is forthcoming in Fredrik Espinoza's PhD thesis, due in the fall of 2002.

[6] http://www.winamp.com.

[7] There are many reasons to use a peer-to-peer solution rather than a server-client solution—decentralization, load-balancing, robustness, scalability, etc.

and a community of users. With Jini, RMI, Sockets, etc., and even plain JXTA, a developer must expend greater effort to achieve the same level of functionality.

What does JXTA add? JXTA adds the plumbing of the peer-to-peer network. We believe JXTA has the potential to become a de facto standard for generic peer-to-peer. At the very least we expect to be able to use JXTA for a number of development iterations of Briefcase Connectivity. As a new and improved version of JXTA is released, we plug it into Briefcase Connectivity and start to benefit from the enhancements. However, the Briefcase Connectivity system is built in such a way that we can use any type of underlying peer-to-peer system with minor alterations of interfacing code [9].

2.1 Design

At the most basic level, our design of Briefcase Connectivity enables instances of the same service residing in different briefcases to exchange messages.

In Fig. 1 we see four briefcases running Briefcase Connectivity. Briefcase *D*, for example, is running three services, *S2*, *S3*, and *S4*. These services use Briefcase Connectivity to communicate with *S2* in Briefcase *A*, *S3* in Briefcase *B* and *C*, and service *S4* found in all the briefcases, respectively.

The following list presents the design criteria for Briefcase Connectivity: the sView network should not rely on a central arbitrator for any significant portion of its operation; the sView network should be dedicated to sView briefcases and their services; a briefcase must be able to discover the network; a briefcase must be able to discover other briefcases on the network; a briefcase should be able to discover a particular briefcase on the network; a briefcase must be able to join/leave the network intermittently; the joining/leaving of briefcases should not disrupt

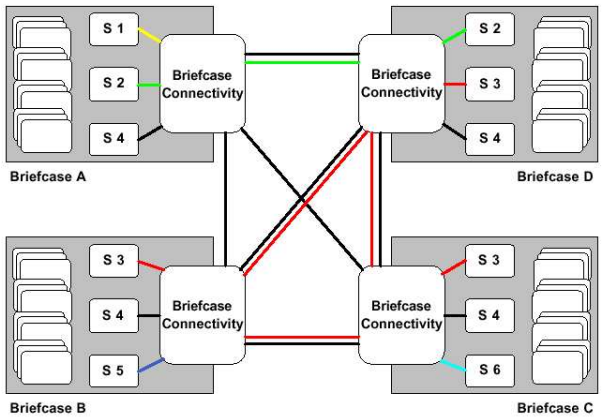

Figure 1. Different instances of the same service communicate via Briefcase Connectivity.

the network; a briefcase must be able to communicate with other briefcases; the communication protocol must carry service-specific protocols; the communication protocol must accommodate a heterogeneous set of service protocols; communication between briefcases should be secure; Briefcase Connectivity should maintain an accurate view of the network; Briefcase Connectivity must make a best effort to deliver messages

We divide the design into three parts: Sect. 2.1.1 describes the network component that enables a briefcase to participate in the network. Section 2.1.2 describes the network interface that is made available to sView services. Finally, Sect. 2.1.3 presents an architectural overview of the design of Briefcase Connectivity.

2.1.1 The Network Component

The sView network is comprised of briefcases upon which sView services that require this network can be deployed. Services that require a sView peer-to-peer network connection with their peers use Briefcase Connectivity as a provider of this function.

We have elected to use JXTA technology as a basis for our design and implementation. The

peer-to-peer protocols [10] implemented by the JXTA reference implementation exceed the requirement criteria listed in the previous section.

Briefcase Peers. Our natural inclination is to represent the sView briefcase as a peer on the JXTA network. To achieve this, the Network Component must implement the JXTA protocols [10] and become a member of a peer group. By default, all peers are members of the "World Peer Group" and, by virtue of the protocols required for inclusion in this group, are capable of discovering resources within the group, including other peer groups, peers, pipes (i.e., connections between peers), etc. The uniform addressing scheme that JXTA provides solves the problem of addressing briefcases. However, we need a way for briefcases to interact with each other.

Peer Interaction. Peers in the JXTA network interact with each other by invoking services. For example, a peer might provide a client-server style service for downloading movies. When a peer wants to make a service available to other peers, it must create a pipe to which clients attach their own pipes. By connecting their pipes to the server pipe, clients can exchange data with the server and access the service.

In sView, we can envision how briefcase peers will interact. Each briefcase runs a messaging service that is equipped with two pipes: one to which other services connect, and the other for connecting to other services. When a briefcase joins the network, it publishes its messaging service advertisement to the peer group. Other briefcases collect these advertisements and extract the enclosed pipe advertisements when they wish to communicate. It is intended that the messages carry sView service specific protocol messages. To eliminate unnecessary overhead, the briefcase messages must be compact.

Briefcase Peer Group. The JXTA peer group abstraction enables us to build a briefcase peer group that is isolated from the rest of the JXTA network. This is advantageous for a number of reasons. First, we can more easily implement trust mechanisms that are valid for the entire group. Second, we can leverage the peer group abstraction to create specialized sub-groups on top of the primary briefcase group. For example, a number of briefcases might deploy an auctioning service that creates a centrally coordinated network, similarly to the Sentinel, as we will see in Sect. 3. Briefcases interested in the service can then join the group and participate in a mediated auction. When the winner of an auction is announced, the buyer and seller may join a private one-on-one network where they conduct a secure transaction over encrypted channels. The peer group abstraction also gives us the flexibility to optimize the sView network's organization. If we discover that a fully decentralized network is sub-optimal, we can incorporate super briefcases (reminiscent of Milgram's highly connected peers) in the network to create a more hierarchical organization and enhance network performance.

2.1.2 The Service Interface

Now that we have a network component capable of carrying messages between briefcases, we need an interface that allows sView services to access the network to communicate. The idea is that any service that requires a network of sView briefcases should not have to design and implement a network architecture or bother with low-level network protocols. Instead, services should be able to access the sView network via a simple interface that Briefcase Connectivity provides. In this sense, Briefcase Connectivity can be seen as an abstraction that provides the API to the sView network protocol. The service interface defines operations for creating and releasing a network connection as well as operations for sending and receiving messages.

Several sView services in one briefcase may use Briefcase Connectivity simultaneously to participate in the network. For this reason, we need a way to deliver incoming messages efficiently and correctly to specific services. As messages arrive from the network, the service interface will deliver the messages to the appropriate service message queue of each service.

The Briefcase Connectivity protocol must be general in order to accommodate any type of service communication requirement, and simple in order to make designing sView network services easy. The operations mentioned above—create, release, send, and receive—suit our criteria of generality and simplicity. When a service creates a connection to the sView network, it receives a handle to a message queue identifiable by the service's name and ID. It receives messages directly from the queue, and sends messages by invoking the send operation defined by the service interface. When the service is finished with its network session, it releases its connection and loses its message queue.

2.1.3 Putting it Together

Combining the network component with the service interface yields Briefcase Connectivity. Our briefcase is represented by a fully qualified JXTA peer that participates in the JXTA network. Briefcases interact with each other by publishing and discovering advertisements for their message services. SView services that wish to communicate over the network must establish a connection to the network, which amounts to obtaining a handle to a private message queue from the service interface. The services can then send messages via the service interface that delivers them to the network component. The network component in turn sends the message on the JXTA network to another briefcase, where it is delivered to the specified recipient service.

2.2 Implementation

The Briefcase Connectivity architecture consists of a peer-to-peer module, a processing and demultiplexing module, service mailboxes, and a simple messaging protocol and message format. When the Briefcase Connectivity service is started in the sView briefcase, it first creates the peer-to-peer module, called JXTAMessenger. JXTAMessenger creates a briefcase peer on the JXTA network. Once the peer is established on the network, the processing and demultiplexing module, called CommManager, is created. This component prepares the internal mechanisms required for formatting and processing messages to and from sView services. When CommManager completes its initialization tasks, Briefcase Connectivity activates both mechanisms. At this point, other sView services may register themselves and receive mailboxes, called ServiceMailboxes, from which they receive messages. At the same time, they may send messages as well as receive network information from Briefcase Connectivity.

Messages arriving at the transport layer are passed up to JXTAMessenger where they are checked for correctness and delivered to CommManager. The messages are parsed, reformatted as SviewMessages, and then delivered to the appropriate ServiceMailbox. Services send messages by delivering them to CommManager where they are formatted and addressed. Once formatted, they are passed down to JXTAMessenger where they are put on the wire.

Please refer to [9] for more details about the implementation of Briefcase Connectivity.

3 Proof of Concept: The Sentinel

Let us consider electronic services in more general terms. Many electronic services demand a high level of user interaction and focus. Some of these services may present information in a graphical user interface, and may require the user

to physically interact with the GUI by making selections or typing information. Other services may prompt the user with visual or audio cues when they have completed a task or require input from the user. ICQ, for example, is an instant messaging application that is infamous for its highly audible ”Uh-Oh!” that sounds when a new message arrives. In an informal setting, such as at home, these distractions are un-intrusive. However, in more formal settings, such as at the office or in a meeting, such distractions are intrusive and potentially disruptive. In a typical meeting, many of those attending have with them mobile phones, PDAs, and laptop computers. A potentially large number of electronic services may be running on these devices, heightening the risk of intrusive events disrupting the flow of the meeting.

The FEEL Project[8] is a cooperative endeavor between SICS, the FUSE group at the Royal Institute of Technology (KTH), and the IAM team at the University of Southampton, and is part of the larger, EU funded program The Disappearing Computer[9]. FEEL focuses on managing the intrusiveness of pervasive and ubiquitous technology. SICS role in the project has been to provide a software platform (sView) for implementing the project’s concepts. FEEL envisions that each member of a planned or ad-hoc meeting has with him one or more computing devices running the sView briefcase. When the group convenes, their briefcases collaborate transparently, determining a level of intrusiveness appropriate for the meeting. Once the intrusiveness level is determined, all services executing in the briefcase are forced to behave in a manner appropriate to the decided upon level. For example, if the group decides upon a low level of intrusiveness, an instant messaging service would not give audio cues, and a mobile phone would redirect calls to voice mail. Briefcase Connectivity’s ad-hoc, peer-to-peer capability tenders the type of core functionality that makes this possible. However, we need a service in the briefcase that carries out this task.

Sentinel (Fig. 2) is a distributed sView service that was developed for the FEEL project. The first implementation of Sentinel uses Briefcase Connectivity to perform a distributed voting task. Users running Sentinel are presented with a GUI that displays the number of active Sentinels, representing the users who are present at the meeting. The user adjusts a slider in the GUI to select his preferred intrusiveness level. Adjusting the slider activates a voting process that is carried out among all of the Sentinels. Each Sentinel presents its vote to the coordinating Sentinel and when the voting process is complete, the GUI displays the average of all the collected votes, the number of voters that participated in the election, and a color-coded panel that indicates whether or not the user’s vote was counted in the most recent election. Based on the vote results, Sentinel regulates how other services interact with the user.

The Sentinel service is fully replicated in each

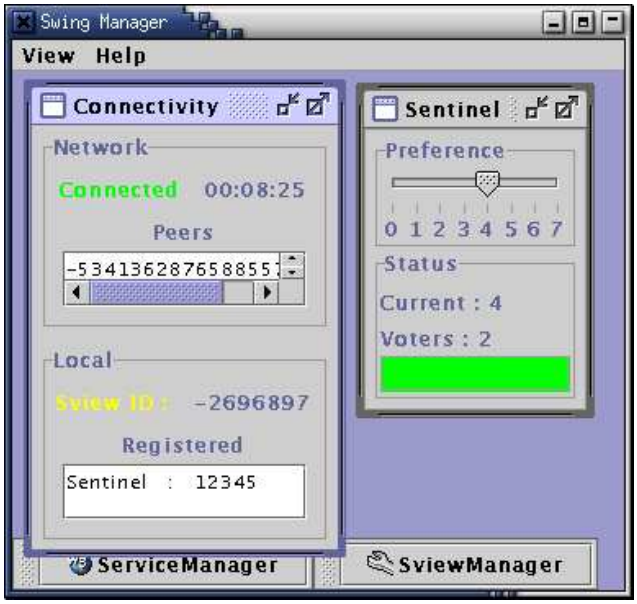

Figure 2. SView running Briefcase Connectivity and Sentinel.

[8]http://www.dsv.su.se/feel.

[9]http://www.disappearing-computer.net.

briefcase. When the voting process begins one of the Sentinels involved will take on the role of coordinator for that particular vote. When the process is finished the result of the vote will be broadcast to all the Sentinels involved and for the next vote it is possible that a different Sentinel will take on the role of coordinator. The fact that any Sentinel can perform the coordination makes the voting system robust. If the current coordinator were to break down another would take its place.

One problem we noticed while working with Sentinel was the timeliness, or rather untimeliness, of some message deliveries. Running a group of Sentinels on a LAN produced no ill effects, but as soon as other Sentinels (which were scattered across the Internet) joined the group, Sentinel's accuracy declined considerably. The JXTA v1.0 Protocols Specification also states:

> "Due to [the] unpredictability of P2P networks, assumptions MUST NOT be made about the time required for a message to reach a destination peer. JXTA protocols SHALL NOT impose any timing requirements for message receipt."

Sentinel employs a timing mechanism that determines the length of the voting process; the fact that JXTA's protocols do not support real-time applications poses a problem for Sentinel. However, JXTA's flexibility allows us to enhance any aspect of the technology, making it possible to design networks that provide real-time functionality. In addition, we can leverage JXTA's peer group facilities to create dedicated Sentinel groups. Within these small peer groups we are able to implement and guarantee a higher quality of service.

Apart from the timeliness of message deliveries, Sentinel is free from problems. The service was easy to implement, due primarily to the simplicity of programming to Briefcase Connectivity's API. At the time of writing, other Briefcase Connectivity services have been implemented, including an on-line help service and an instant messaging service.

4 Related Work

Today's peer-to-peer systems can be classified into three broad categories: centrally coordinated, hierarchical, and decentralized.

Centrally coordinated systems operate in a fashion similar to client-server systems. A central server coordinates the activity of the peers in the network and mediates information that the peers may later act on, for example, to contact each other. In hierarchical systems, the responsibilities of a central coordinator are delegated to a tree of coordinators around which peers form themselves into groups. Of the three categories, decentralized peer-to-peer has drawn the most interest, due to the fact that these systems are robust, scalable, and virtually self-sustaining—characteristics desirable in distributed applications. Four notable decentralized peer-to-peer systems are Gnutella[10], Freenet[11], OceanStore[12], and Project JXTA.

Gnutella. Gnutella is a protocol for distributed information searching and sharing, implemented primarily by file sharing applications. The Gnutella protocol defines the way in which servents[13] communicate over the network. Gnutella is currently the largest peer-to-peer network in use, boasting millions of users.

Freenet. Freenet is an anonymous information storage and retrieval system. This description is a bit of a misnomer, however, as Freenet does not guarantee permanent storage. The goals of the Freenet project are to provide a system that guarantees anonymity

[10] http://gnutella.wego.com

[11] http://freenet.sourceforge.net

[12] http://oceanstore.cs.berkeley.edu/

[13] A Gnutella peer is called a servent because it acts as both a SERver and a cliENT.

for both producers and consumers of information, thwarts third party attempts to deny access to information, and provides efficient storage and routing of information.

OceanStore. OceanStore is a wide-area peer-to-peer storage architecture [2]. Currently under development, OceanStore is intended to be a self-sufficient, globally spanning archival system that links together servers from around the world. In return for a nominal fee proportional to the amount of desired storage space, an OceanStore user has access to persistent data—personal files, configuration data for a personal computing device, etc.—from anywhere in the world.

Project JXTA. Project JXTA is an ongoing, monolithic open-source effort bent on improving modern distributed computing, especially in the area of peer-to-peer [7, 11]. JXTA's developers claim that the Internet's three fundamental assets—information, bandwidth, and computing power—are underutilized due to the prevalence of the client-server model. The project's design objectives are derived from what the JXTA designers consider to be shortcomings of existing peer-to-peer systems: the inability to inter-operate, platform dependence, and the lack of true ubiquitous support for heterogeneous devices. At the highest level of abstraction, JXTA is a set of protocol specifications that define the basic activities of any conceivable peer in a peer-to-peer network. It is intended that this will provide a flexible framework in which to design nearly any type of peer-to-peer system.

The common ground shared by all of these systems is threefold:

1. They embrace decentralization but do not necessarily exclude centralization. All of these systems attempt to minimize their reliance on central points that provide functionality essential to the workings of the system such as processing, address resolution, arbitration, etc.

2. They leverage resources—processing power, storage space, bandwidth, digital content, human presence, etc.—that would otherwise be underused or unused

3. They tolerate and even expect dynamic connectivity

Briefcase Connectivity is similar to Gnutella in that it is decentralized and nodes act as servers and clients. The main difference is that Briefcase Connectivity was designed to be a generic peer-to-peer system on top of which service specific protocols can be placed. This allows us to use the communication layer for more than file sharing. Freenet and OceanStore focus on providing reliable storage of data and files within the peer-to-peer community—Briefcase Connectivity is focused on communication between peers. When peers can communicate using any chosen protocol they can also establish service specific protocols and infrastructure in parallel. Thus Briefcase Connectivity is more general and could, in theory, implement the functionality of both Freenet and OceanStore, via separate services. JXTA is most similar to Briefcase Connectivity, and rightly so, since Briefcase Connectivity builds on JXTA. The difference is that Briefcase Connectivity is designed to work at a higher level of abstraction, providing peer-to-peer capabilities to other services in the sView briefcase.

5 Conclusions

We have presented the development of Briefcase Connectivity. To design a generic peer-to-peer service like Briefcase Connectivity, we needed to understand the field of peer-to-peer.

Our analysis of this computing paradigm[14] taught us that it is powerful and flexible, but also that its benefits are not easily won.

Although it is relatively new, JXTA technology offers designers a powerful and flexible medium in which to build peer-to-peer systems. Our own design and implementation attests to this fact. Aided by JXTA, Briefcase Connectivity enables any service to access the sView network and communicate with other services. We hope that the success of Briefcase Connectivity and services like Sentinel generates an "sView network effect" and inspires service providers to build new and exciting services for the sView platform.

We intend to use Briefcase Connectivity as a testing ground for future research. This research includes, but is not limited to: analyzing JXTA's performance and optimizing Briefcase Connectivity; implementing an accountability mechanism to ensure the equal and proper usage of sView network resources; deploying a reputation system on the sView network to spin a "web of trust"; fine-tuning Sentinel for the FEEL Project; and realizing our ideas for sharing individually created services between users.

6 Acknowledgments

The work described in this paper was partially financed through the FEEL EU project and the sView SITI project[15]. The authors wish to thank the rest of the OASIS team at SICS, including Markus Bylund, Ola Hamfors, Anna Sandin, and Stina Nylander.

References

[1] *Nexus, Small Worlds and the Groundbreaking Science of Networks*. W. W. Norton & Company, 2002.

[2] David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Christopher Wells, Ben Zhao, and John Kubiatowicz. OceanStore: An Extremely Wide-Area Storage System. Technical Report UCB/CSD-00-1102, University of California at Berkeley, Computer Science Division, Berkeley, California 94720, May 1999.

[3] Markus Bylund. Personal Service Environments - Openness and User Control in User-Service Interaction. Licentiate of Philosophy Thesis, Uppsala University, 2001.

[4] Markus Bylund and Fredrik Espinoza. sView – Personalized Service Interaction. In Jeffrey Bradshaw and Geoff Arnold, editors, *Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 2000)*, pages 215–218, Manchester, UK, April 2000. The Practical Application Company Ltd.

[5] Markus Bylund and Annika Wærn. Personal Service Environments - Openness and User Control in User-Service Interaction. Technical Report T2001:07, Swedish Institute of Computer Science, Kista, Sweden, May 2001.

[6] Fredrik Espinoza and Ola Hamfors. ServiceDesigner: A Tool to Help End-Users Become Individual Service Providers. In *Proceedings of HICSS-36, Hawaii International Conference on System Sciences*, January 2003. Forthcoming.

[7] Li Gong. Project JXTA: A Technology Overview. Web, 2001.

[8] Ola Hamfors. Service Designer - Lets the End-user Create her Own User-Interface to

[14] This analysis was more extensive than is apparent in this paper. Please see [9] for more details.

[15] http://sview.sics.se.

Web Services. Master's thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

[9] Lucas Hinz. Peer-to-Peer Support in a Personal Service Environment. Master's thesis, Uppsala University, Uppsala, Sweden, 2002.

[10] JXTA v1.0 Protocols Specification. Web, 2002.

[11] Project JXTA: An Open, Innovative Collaboration. Web, 2002.

[12] Carl Shapiro and Hal R. Varian. *Information Rules: A Strategic Guide to the Network Economy*. Harvard Business School Press, Boston, Massachusetts, 1999.

Chapter 11

Paper E: Towards Individual Service Provisioning

Towards Individual Service Provisioning

Fredrik Espinoza
Swedish Institute of Computer Science
Box 1263
164 29 Kista, Sweden
+46 8 633 1500
espinoza@sics.se

ABSTRACT
With the emergence of modularized component-based electronic services, such as Web Services and semantically tagged services, *Individual Service Provisioning*, wherein any user can be a service provider, can become a reality. We argue that there are three basic requirements for such an architecture: a personal service platform for using services, tools for creating services, and a network for sharing services, and we present our motivation, design, and implementation of these parts. With our enabling architecture we hope to demonstrate a feasible prototype system that stimulates the emergence of more specialized services for all users.

Categories and Subject Descriptors
H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces – *collaborative computing, asynchronous interaction, computer-supported cooperative work, synchronous interaction, web-based interaction*

General Terms
Management, Design.

Keywords
Individual service provisioning, service composition, peer-to-peer, service sharing

Copyright is held by the author/owner(s).
IUI'03, January 12–15, 2003, Miami, Florida, USA.
ACM 1-58113-586-6/03/0001.

1. INTRODUCTION

The Internet is becoming more service centered. Web sites offer interactive services as an alternative to the plain publishing of information: databases may be searched, goods may be purchased, and e-mail may be sent, etc. With the emerging Semantic Web [1], content providers can code web content semantically and service developers can harvest the content with new semantically enabled services.

In parallel, other networks are beginning, or continue, to offer services as an added value to customers. The telecom operators have featured services such as voice mail and short message services (SMS) for some time, and the new broadband Internet Service Providers (ISP) are starting to offer interactive on-demand services akin to the digital cable and television networks' interactive media services. On the home front, home area networks offer users the option of controlling home entertainment systems, alarms, or major appliances (for example, Electrolux's screen fridge[1], and E2 Home's appliance control system[2]). Web Services, another recent Internet development, are set to power systems such as these and others, behind the scenes.

With the emergence of more modularized component based services, *Individual Service Provisioning* can become a reality—and any user can be a service provider. We suggest that there

[1] http://www.electrolux.com/screenfridge/

[2] http://www.e2-home.com/

are three basic requirements for individual service provisioning:

- A *Personal Service Environment* (PSE) for accessing, storing, and using services. With a PSE the user gets a coherent and unifying service experience which fosters positive feedback in terms of increased usage and numbers of users, and, as a result, greater incentive for developers to provide services. Without a PSE, the user's service experience is bound to be fragmented, which will lead to the opposite effects
- Tools to create services. Traditional service development tools will not suffice if the aim is to enable ordinary end-users to become service providers; the tools have to be simple and intuitive. If this comes at the expense of limiting the possible complexity of produced services, so be it. The most important issue is to encourage service production. More services will lead to more incentive to use the overall system, which in turn leads to more users, which in turn gives professional developers the incentive to produce the more complex services
- A network within which to share services. As users and professional developers provide services they must be made accessible within the user community; and the means for this should be built into the PSE from the start. This is especially important for user-constructed services, as these otherwise are difficult to find

In this paper we describe these three components, as they have been designed and implemented in our prototype system. We try to demonstrate a viable and feasible system that can stimulate individual users and professional service providers to create more specialized services for all users.

2. BACKGROUND

This work is grounded in a vision of the user in the center of an electronic service world. In this world, the user leaves behind the desktop computer with its set of applications, as the services are brought along, virtually, into any and everyday situations. Services are personal and specific to the user's needs, and the right service is available at the right time. The vision does not rely on artificial intelligence as an enabling technology, but rather on simple computational constructs combined with the input of human intelligence.

The Internet is a carrier and a catalyst in this vision. The World Wide Web has provided an important growth of computer literacy, content provision, and technological innovation, as well as the necessary basic building blocks in the form of protocols and development tools. E-mail, instant messaging, peer-to-peer systems, among others, have also contributed to the overall growth and development of the Internet and its community.

Now we are entering a second stage of the Internet evolution. While the first was driven by human communication, the second stage is driven by our wish to enable *machines* to communicate:

- Web Services are programmatically accessible functional components, made accessible over the common infrastructure of the web. They provide a unified interface to distributed functions that can be combined and integrated into applications and end-user services
- The technology of the Semantic Web enables resources to be semantically coded. The resources can be web sites containing information, web services that provide function, or physical devices that can act in the physical world. The semantic coding allows programs to understand the purpose and function of resources as well as the relationships between the concepts describing them
- Ubiquitous computing, as originally described by Weiser [6], will make computers

disappear, as they sink into the fabric of everyday life. Computing devices will be embedded in everyday objects and they will all be connected to provide emergent functionality—this requires new network and hardware technologies and software architectures

Of course, there is little point in making machines communicate unless humans can benefit, and, obviously, the desire for communicating machines is grounded in a longing for more effective, functional, and usable services for humans. With the new technologies of the Internet in combination with new hardware devices and infrastructure the right tools will be available for developers and content providers to create an abundance of services. End-users, however, can also be service providers.

3. SVIEW: A PERSONAL SERVICE ENVIRONMENT

To enable end-users to be service providers we first need a platform to use the services. The state of the art in generic service platforms is, at the moment, the web. The current solutions are, however, not optimized for the users. Heavy bandwidth requirements is one problem: web based services are fine for using over a faultless high-speed connection in an office, but users barely get by using a modem connection at home. Another problem concerns the multitude of platforms and proprietary solutions required to access the typical range of services. These may be acceptable when the total number of services is relatively low. But when the numbers grow, so does the inevitability of a hopeless breakdown of the user situation. There is, or will soon be, a need for another solution.

In previous work, we (Open and Adaptive Service Infrastructures (OASIS) group at Swedish Institute of Computer Science) have designed and built the *sView Personal Service Environment* [2] (http://sview.sics.se). In sView, each user has a personal service briefcase in which to store and execute his or her services. When the user's briefcase is running locally, on the user's own computer, the user may interact with the services using powerful graphical user interfaces; each service is then represented by a window with full interaction capabilities. When the user logs out, the briefcase and the services within are stopped and serialized, and synchronized to a centrally located (multi-user) Enterprise server, where they are reinstated and continue to execute. From the Enterprise server users can reach services using web, WAP (mobile phone), or Short Message Service (SMS) based interfaces optionally provided by each service. Thus the briefcase is continuously active, the user's session is never stopped, and services always run.

While the sView environment supports the user in interacting with electronic services, the *ServiceDesigner* supports the user in creating those services.

4. SERVICEDESIGNER: A TOOL FOR CREATING YOUR OWN SERVICES

Web services, as part of their design, have no user interface. They are highly useful to application developers and business-to-business architects but for the end-user a user interface shell must be constructed and applied to the complete composition. This is the strength and weakness of web services.

To counteract this, and to empower the user, we have created the ServiceDesigner—an sView service for direct access to web services within sView. With the experience gained from the work with sView, we see web services as an opportunity to users as well as to developers.

In ServiceDesigner [3], a user can enter the URL of any WSDL compliant web service, after which a graphical user interface is automatically generated, complete with the necessary logic to use the service (The Apache FAQ[3] recommends ServiceDesigner as a tool to test web services). ServiceDesigner also includes a visual editing

[3] http://xml.apache.org/soap/faq/faq_chawke.html

tool with which the user can modify the interface to his or her liking. When the user is satisfied with the interface design, and having tested the function of the service directly in the interface, a fully compatible sView service can then be generated in sView's standard JAR file format.

But ServiceDesigner also allows the user to combine several web services to create a composite service. By connecting input and output parameters in a visual editor, a user can create a flow of data between any numbers of web services. Combined with built-in functions for simple arithmetic, repeaters, and timers, arbitrarily complex combinations can be produced. And, similarly to the simple case of using one web service, the resulting service composition can be immediately tested using the generated interface, and the final service can be generated into a fully sView compliant service.

With sView and ServiceDesigner a user is able to access a personal set of services in his or her personal briefcase and also create appropriate services to fill individual needs. The ServiceDesigner is relatively easy to use, even for inexperienced users, but in the optimal situation it would be even better if services already existed. Given that one user finds the need for a particular service one could argue that there could be other users with similar needs. If a user could find and download an existing service there would be no need to expend the effort to create the service. Therefore, the third necessary component is a community of interconnected users.

5. BRIEFCASE CONNECTIVITY: A GENERIC PEER-TO-PEER SYSTEM TO ENABLE SHARING OF SERVICES

The sView service *Briefcase Connectivity* provides the capability for other services to connect to a sView peer-to-peer network [4]. Based on Project JXTA (www.jxta.org), Briefcase Connectivity solves two problems in sView: first, it enables briefcases to find and connect to each other to form a peer-to-peer network. Second, it allows services in sView to communicate with one another by message passing.

The sharing of services is straightforward and reminiscent of similar file sharing systems (Napster, Gnutella, etc.). Users create services with ServiceDesigner, and as they do, keywords described in WSDL documents for the included parts, are gathered and embedded in the final service. The user is also able to manually tag the finished service with additional appropriate keywords. The keywords are used when indexing the user's created services, which are published in the user's list of available services. Other user's can search through the complete peer-to-peer network for services fitting the requested keywords. To use a service it is transferred over the network from the publishing user to the requesting user where it is stored in the user's service store.

The sView community can thus be leveraged to make more services available to users.

6. INDIVIDUAL SERVICE PROVISIONING

Just as anyone can publish a web page about himself or his interests or business, anyone should be able to publish services for others to use. With our tools, users are able to create their own services, use them in their personal service environment, and with the Briefcase Connectivity infrastructure, share them with others.

With Individual Service Provisioning we hope to see more specialized services become available to users. By providing the necessary tools and infrastructure, we also hope to achieve a network effect. When users contribute services to the total system in a way that benefits the whole community, the network collection of services increases in size, and its value to the individual also increases [5].

7. ACKNOWLEDGMENTS

We wish to acknowledge the rest of the OASIS group at Swedish Institute of Computer Science,

including Ola Hamfors, Lucas Hinz, Markus Bylund, and Anna Sandin.

8. REFERENCES

[1] Berners-Lee, T., Hendler, J., and Lassila, O., The Semantic Web. *Scientific American*, 284(5): 34-43, 2001.

[2] Bylund, M. and Espinoza, F. sView - Personal Service Interaction. In *Proceedings of the 5th International Conference on The Practical Applications of Intelligent Agents and Multi-Agent Technology*, PAAM 2000, pages 215-218. The Practical Application Company Ltd., April 2000.

[3] Espinoza, F. and Hamfors, O. ServiceDesigner: A Tool to Help End-Users Become Individual Service Providers. In *The Proceedings of the Hawai'i International Conference on System Sciences*, January 6 - 9, 2003, Big Island, Hawaii, 2003. forthcoming.

[4] Fredrik Espinoza and Lucas Hinz. Generic Peer-to-Peer Support for a Personal Service Platform. In *Proceedings of The 2003 International Symposium on Applications and the Internet* (Saint 2003), January 2003. forthcoming.

[5] Katz, M. and Shapiro, C., Systems Competition and Network Effects. *Journal of Economic Perspectives*, 8(2):93-115, 1994.

[6] Weiser, M. The Computer for the Twenty-First Century. *Scientific American*, 265(3): 94-104, September 1991

Appendices

Appendix A

ServiceDesigner Tutorial

Introduction

This tutorial describes the ServiceDesigner sView service, and the service location web site Salcentral.com. This version of the tutorial is based on an older version by Andreas Espinoza.

ServiceDesigner is used for two purposes: to test web services and to create new sView services. A web service is an electronic service that makes use of the Internet as an information source and transportation tool. Web services are described in XML documents using the WSDL (Web Service Description Language).

The Salcentral.com web site is used in this tutorial as an example of a site where you can find WSDL documents that describe the properties and functions of web services. You can use any such site as a source for your WSDL descriptions (for example xmethods.com). You can even use ServiceDesigner to test your own web services as long as they are described by WSDL documents. From the user's point of view, the WSDL document does not contain the actual service itself, but the information (links to software code, etc) necessary to create a service using a tool like the ServiceDesigner. The ServiceDesigner makes use of the information it collects from Salcentral.com to create the service and its interface.

To use ServiceDesigner you first need to download, install, and run sView. You can get the current sView distribution from sview.sics.se. Next, you need to load and run ServiceDesigner.

Loading and Running the ServiceDesigner

After starting sView, make sure the ServiceManager is running. Then select "Use" on the available loader, select the ServiceDesigner from the list of available services, and then press the Load button to load the ServiceDesigner

(the JAR file is downloaded from the sView web site). The "WSDL URL" field is where you enter URLs to WSDL documents that describe web services (see Fig. A.1).

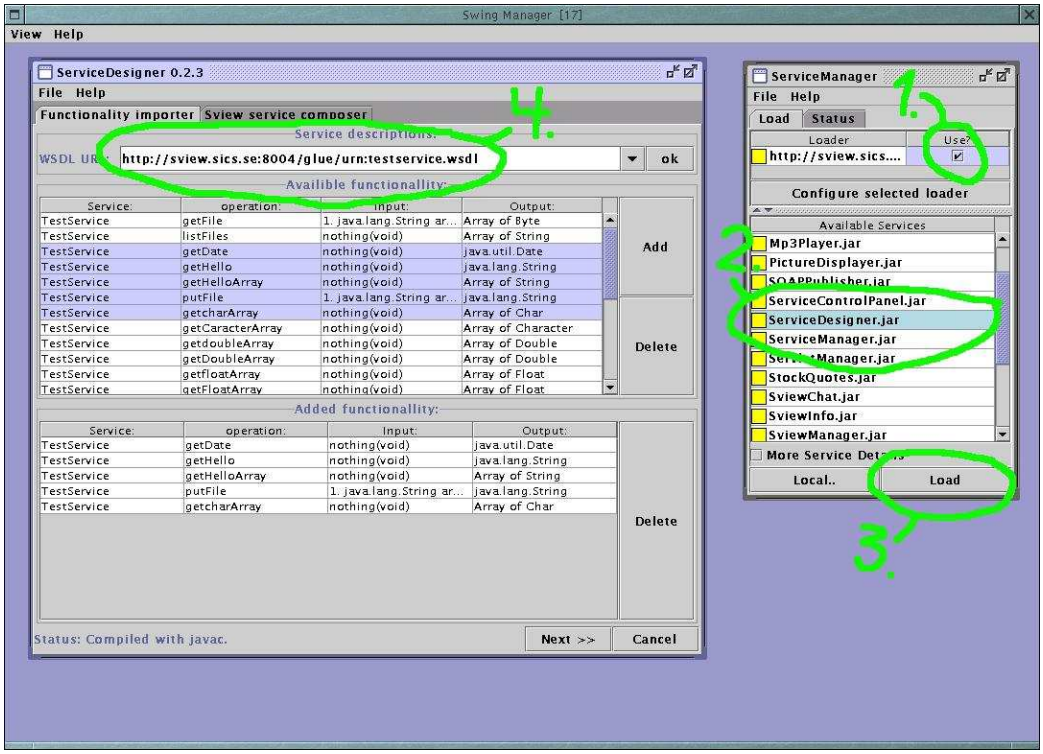

Figure A.1: Running the ServiceDesigner in sView.

Using the ServiceDesigner

The following figures will further clarify, what is stored at Salcentral, what the ServiceDesigner makes use of from Salcentral, and the purpose and function of the ServiceDesigner. The process of creating a service with ServiceDesigner is described in the following nine steps.

1. Get URL to WSDL document for web service to use

The information necessary to create and use a web service is available on the Internet. Currently there is a website called Salcentral.com that collects numerous web services. The site has a search engine allowing the user to search for a service by entering key words (Fig. A.2).

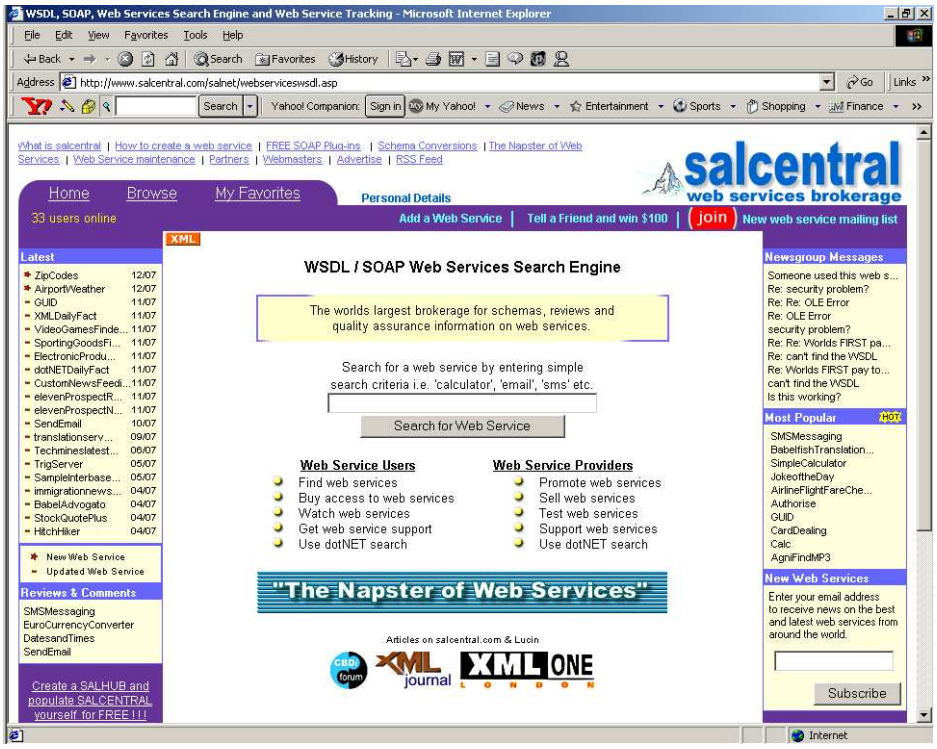

Figure A.2: The salcentral.com Web Services repository site.

In the following example (Fig. A.3) the user has entered a search for the word "language" into the search engine. The results include several language translation services. The user decides to use one of these services. The information needed to create the service is called a "schema." The "Schema location" link takes the user to where the service information (WSDL document) is located.

To the user, the "schema" containing the service information is complicated code written in XML (Fig. A.4). Luckily, the user will not need to decipher the information. The user simply highlights and copies the URL, indicating the location of the "schema."

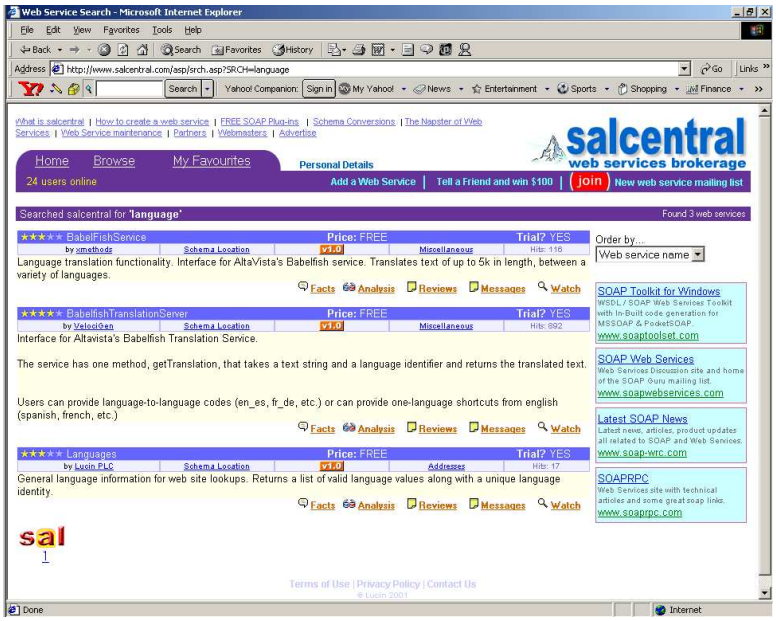

Figure A.3: Examining a service at salcentral.com.

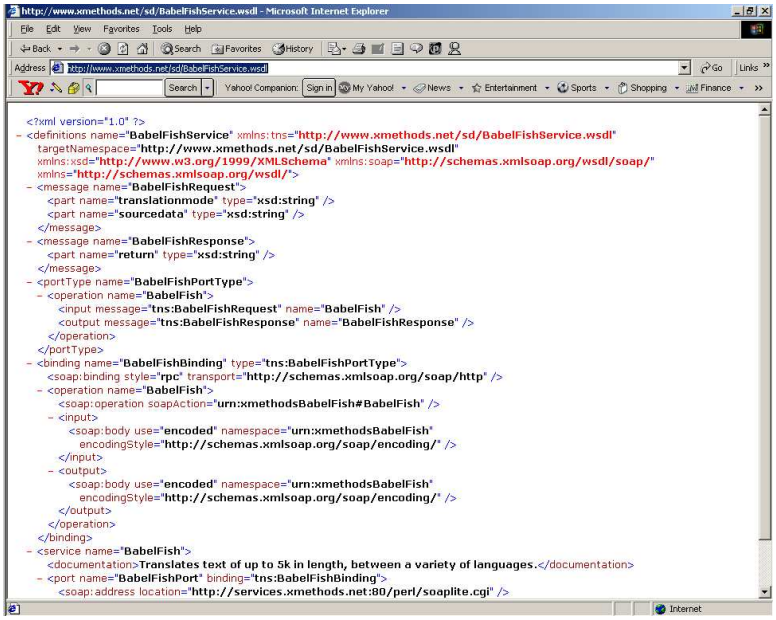

Figure A.4: An example schema (WSDL) for a web service.

2. Enter URL into ServiceDesigner

The user copies the schema location URL and pastes it in the URL field (Fig. A.5). When ServiceDesigner is launched, the URL field contains a default value that you can use for testing, but in this figure, it has been filled with the URL for the language translation schema.

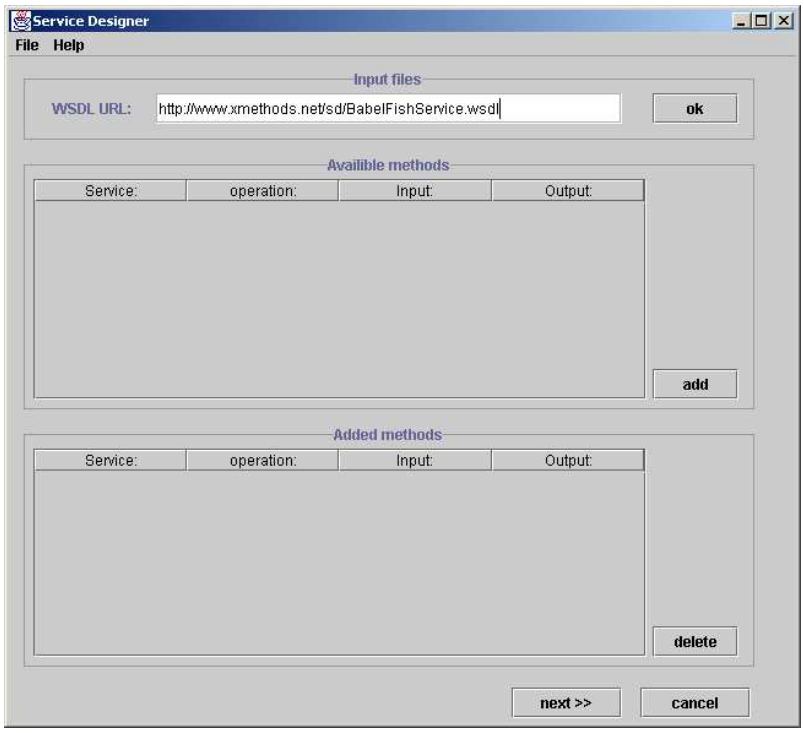

Figure A.5: Pasting a URL into ServiceDesigner.

3. Load WSDL Document into ServiceDesigner

When the "ok" button is clicked, the ServiceDesigner presents the different tasks (methods) the particular service can produce (Fig. A.6). There may be several selectable options in this "Available methods" box. In this case, there is only one option. In this example, BabelFish is a given name for a service and does not explain what the service does. Future versions of the ServiceDesigner will more clearly show what each operation of a service is.

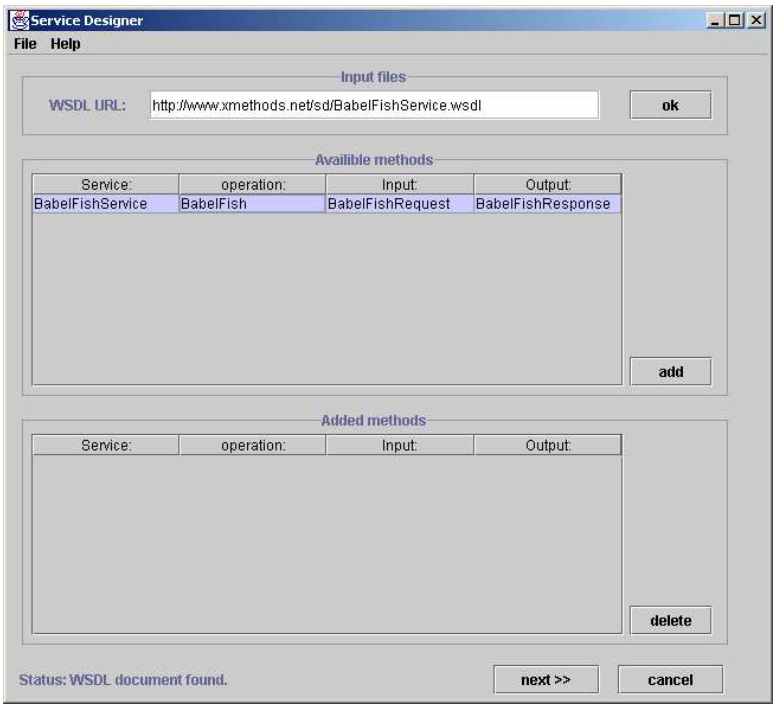

Figure A.6: Viewing the available functional components (methods) of the web service.

4. Select Methods to Use

One task is selected and added (by clicking the "add" button) to the "Added methods" box (Fig. A.7). This box indicates which methods have been selected as "ingredients" for the particular service that is to be created. Now the "next>>" button is clicked and the service is created. An interface is automatically generated and the result can be seen in the next image. Along with the interface, the necessary code for using the service is also created.

5. Test the Service

To test the service, fill in the parameters and right-click the button and choose activate (Fig. A.8). In this case, the translation mode (language to be translated) is entered along with the text data that we want to translate. Clicking the "BabelFish" button activates the service, and the results are displayed in the "Output" box.

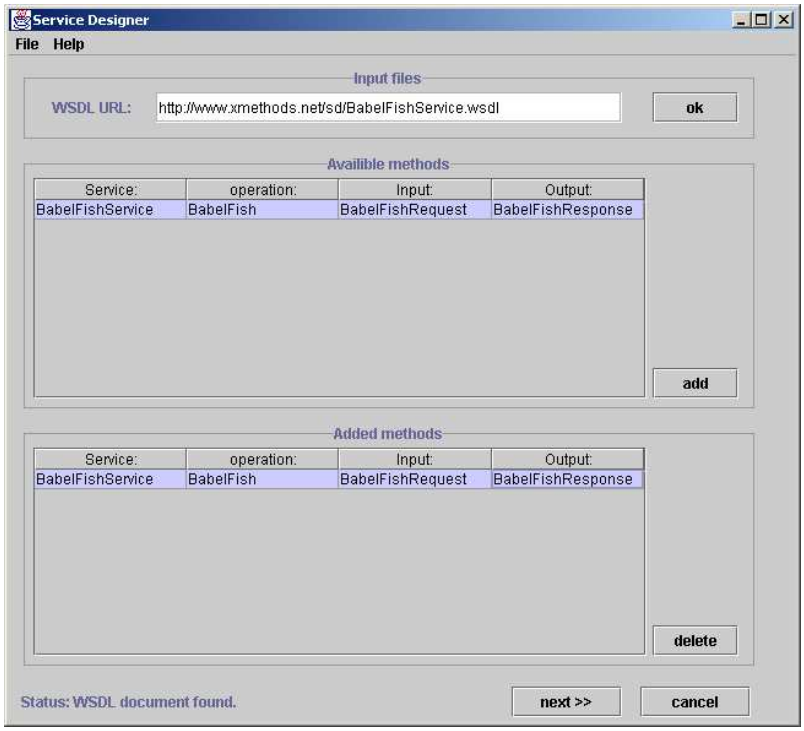

Figure A.7: Selecting which functional components to use.

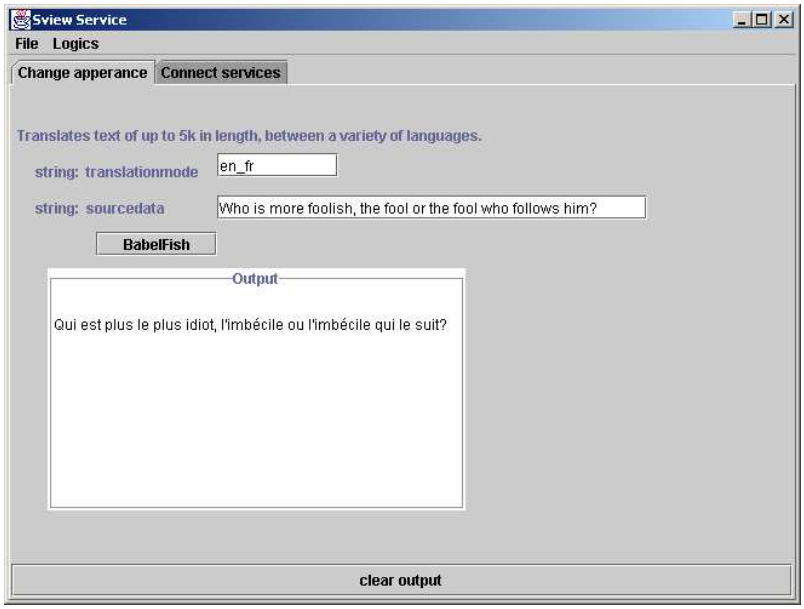

Figure A.8: Testing the service.

The current version of the ServiceDesigner does not prompt the user as to what information needs to be entered in the different fields. This information should be gathered from the information in the web page describing the service. Some indication of what is needed is given by the auto-generated labels.

6. Customize the Interface

The auto-generated interface may be customized. You can right click any component to customize that component. For example, when you right click a label you are able to modify it. If you leave it blank, it disappears. You can move components around by dragging them. You can change the overall size of the service interface by changing the size of the service window.

7. Loading Several Services

If we were to return to Salcentral.com and collect a URL for an additional service, and load its WSDL document as described above, we would end up with several available methods to select from when creating our service (Fig. A.9). We add the ones we want into our "Added methods" box and create the new service by selecting the "Next>>" button.

In the following figure (Fig. A.10), the new service (Shakespearean insult) and the old language service share the same interface screen, but still perform their tasks separately. As we can see, when the "GetShake.." button is activated, the Shakespearean insult service produces an insult in the "Output" box.

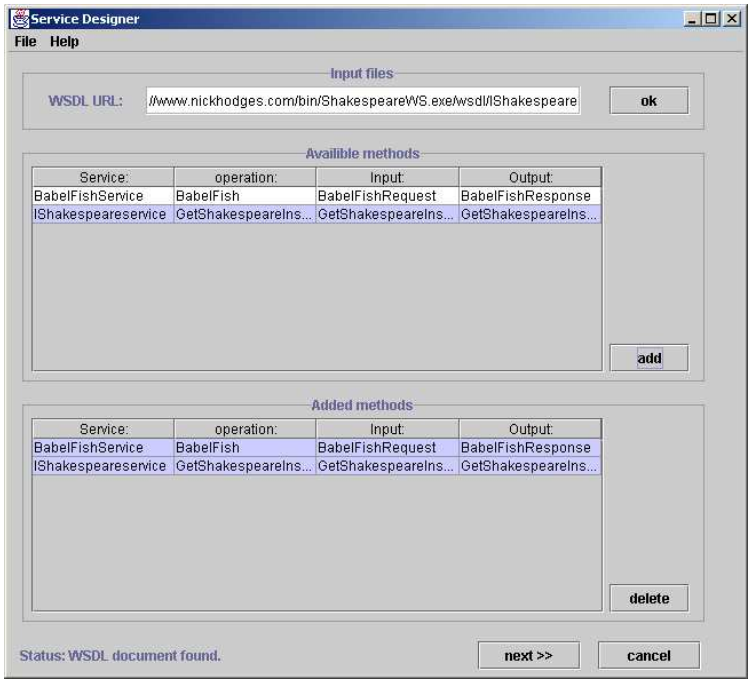

Figure A.9: Selecting from several services.

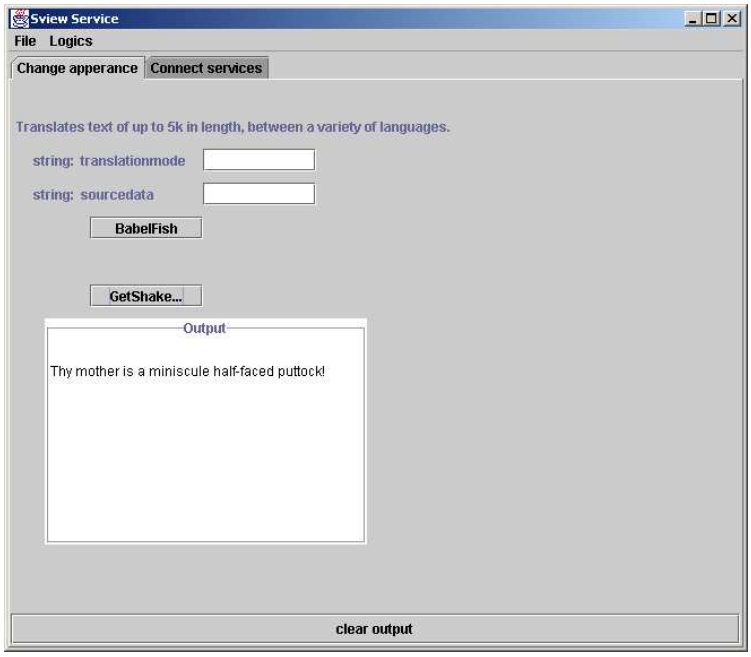

Figure A.10: Two functional components in the same interface.

8. Combining Services

By selecting the "Connect services" tab near the top of the screen, the user is presented with a service connection screen (Fig. A.11). Each service is represented by a box.

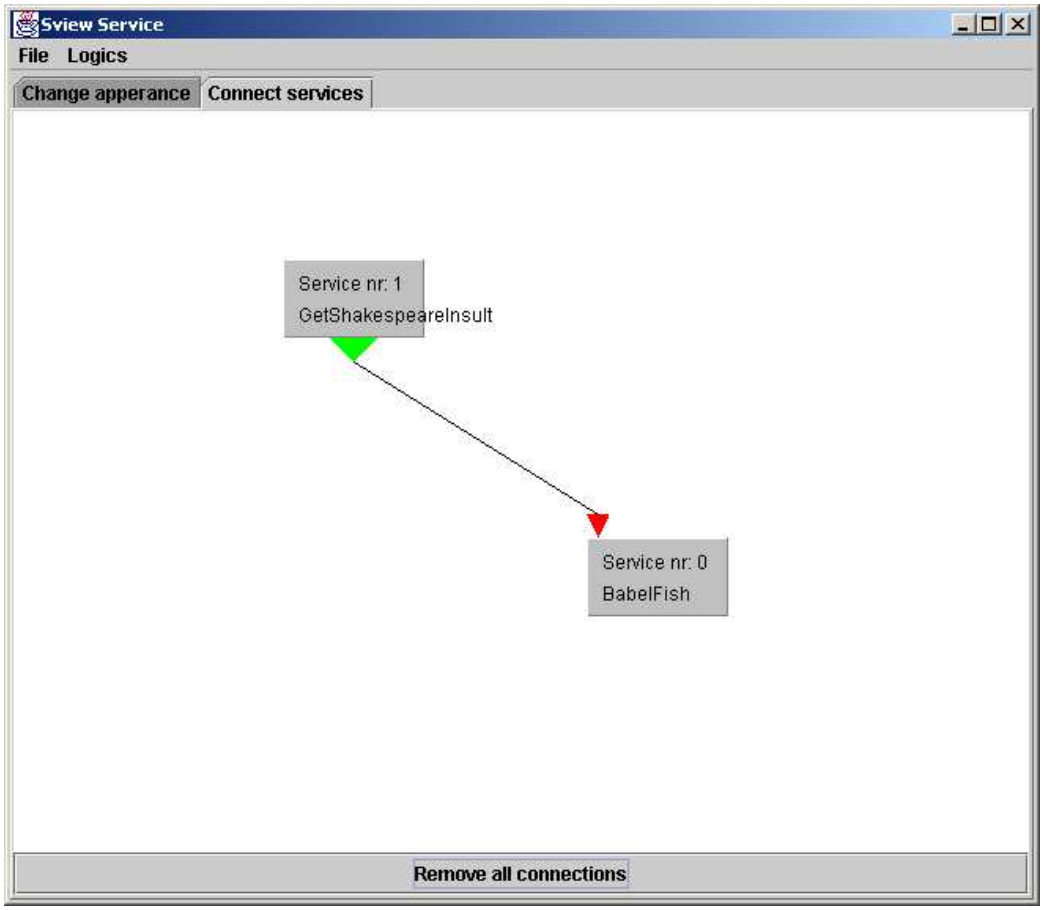

Figure A.11: Connecting services.

Here, the user may connect the services visually, by dragging arrows between them. This indicates the direction, source, and destination of the information to be exchanged. Press <Shift> and left click a service box to drag a connection to another service.

Some operations that you may want to use are very simple, and as such, the overhead that is incurred by using them over the Internet has prompted us to include them locally. They are available in the "Logics" menu. We currently provide the four basic arithmetic operations and a sim-

ple timer/repeater. When you choose one of them from the menu, a corresponding box is added to the connection layout (Fig. A.12).

Figure A.12: Adding local functional components.

The arithmetic operations are used in the same way as web services. The timer/repeater is usually placed as the final box in the connection chain to make the whole operation repeat with a certain time delay. To set the delay, right click the repeater box and choose preferences (Fig. A.13).

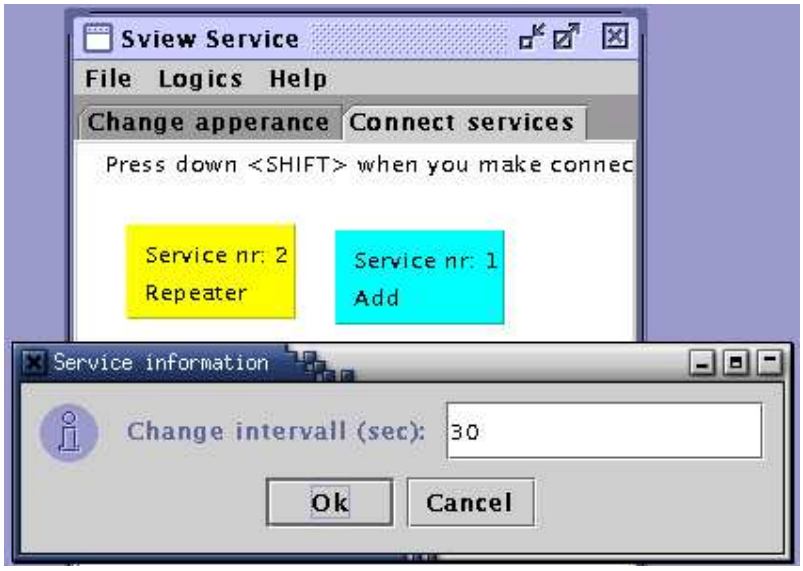

Figure A.13: Setting the delay of the Repeater.

As shown previously, the Translator service requires two pieces of data to perform its service. When making the connection from the Shakespeare service to the Translator service, the user must indicate where the output information from the Shakespeare service should be entered into the input fields of the translation service (Fig. A.14).

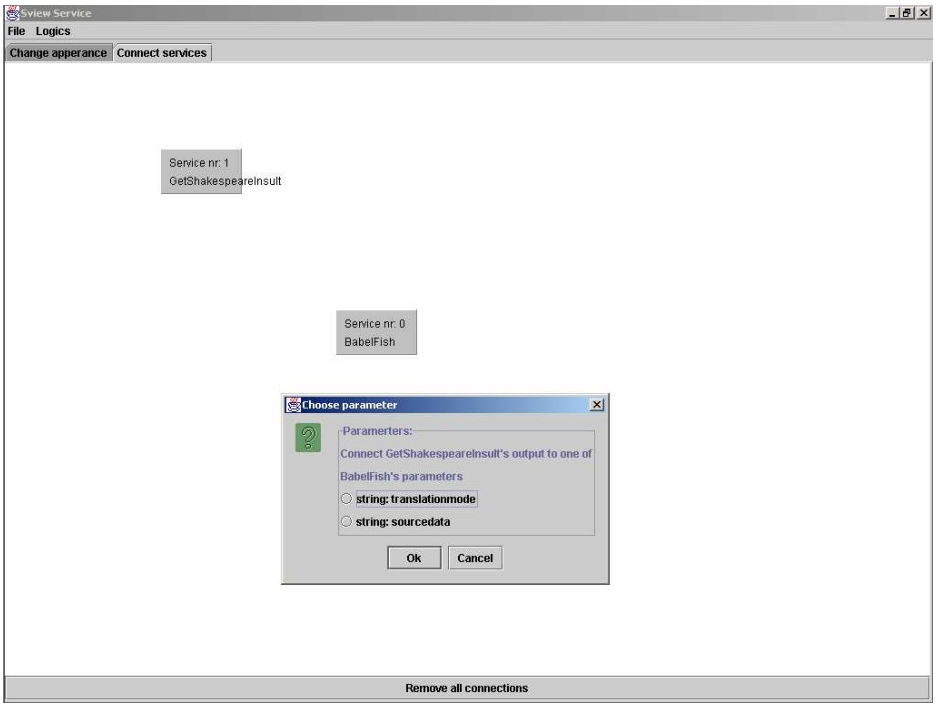

Figure A.14: Choosing where data should go.

In this case, the options are that the incoming data is either the language protocol (en_fr), or the text itself that is to be translated. The correct choice in this case is "Text to be translated," allowing incoming text from "..insult" to be translated. The completed service combination is viewed by going back to the Change Appearance screen (clicking the tab). The interface has now been reduced to only one activation button, and one field where the user shall indicate the language to be translated (Fig. A.15). The result: The new service produces a Shakespearean insult in French or whatever language that was chosen. You can now fine-tune the interface as described above.

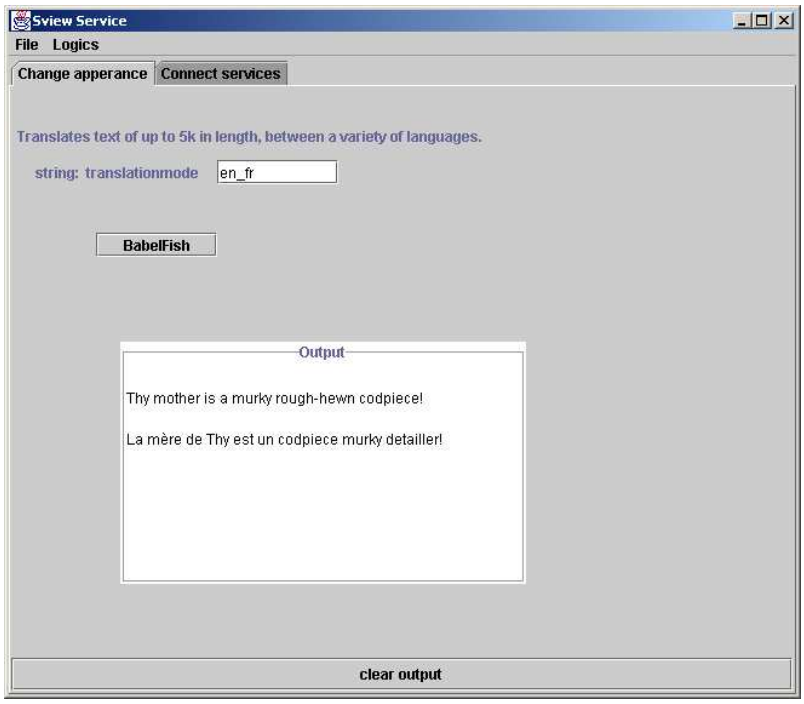

Figure A.15: New interface of the completed service combination.

9. Generate a Real sView Service

When you are happy with your new service, you may generate a real sView service. From the menu bar at the top of the service window, choose "File/Generate Code" (Fig. A.16).

Figure A.16: Generating the sView service.

You are asked to choose a name for your new service. Enter the name and some values that tell others about your service (Fig. A.17). When you press "OK," the service will be generated and compiled and a JAR file will be created. You can find the JAR file of the service in the "services" directory in the root of the sView distribution. The service will also be loaded into sView. You can now give the service JAR file to your friends and they can load it into sView and use it without worrying about what goes on behind the scenes.

Figure A.17: Choosing a name and setting keywords, etc.

Swedish Institute of Computer Science
SICS Dissertation Series

01: Bogumil Hausman, *Pruning and Speculative Work in OR-Parallel PROLOG*, 1990.
02: Mats Carlsson, *Design and Implementation of an OR-Parallel Prolog Engine*, 1990.
03: Nabiel A. Elshiewy, *Robust Coordinated Reactive Computing in SANDRA*, 1990.
04: Dan Sahlin, *An Automatic Partial Evaluator for Full Prolog*, 1991.
05: Hans A. Hansson, *Time and Probability in Formal Design of Distributed Systems*, 1991.
06: Peter Sjödin, *From LOTOS Specifications to Distributed Implementations*, 1991.
07: Roland Karlsson, *A High Performance OR-parallel Prolog System*, 1992.
08: Erik Hagersten, *Toward Scalable Cache Only Memory Architectures*, 1992.
09: Lars-Henrik Eriksson, *Finitary Partial Inductive Definitions and General Logic*, 1993.
10: Mats Björkman, *Architectures for High Performance Communication*, 1993.
11: Stephen Pink, *Measurement, Implementation, and Optimization of Internet Protocols*, 1993.
12: Martin Aronsson, *GCLA. The Design, Use, and Implementation of a Program Development System*, 1993.
13: Christer Samuelsson, *Fast Natural-Language Parsing Using Explanation-Based Learning*, 1994.
14: Sverker Jansson, *AKL - - A Multiparadigm Programming Language*, 1994.
15: Fredrik Orava, *On the Formal Analysis of Telecommunication Protocols*, 1994.
16: Torbjörn Keisu, *Tree Constraints*, 1994.
17: Olof Hagsand, *Computer and Communication Support for Interactive Distributed Applications*, 1995.
18: Björn Carlsson, *Compiling and Executing Finite Domain Constraints*, 1995.
19: Per Kreuger, *Computational Issues in Calculi of Partial Inductive Definitions*, 1995.
20: Annika Wærn, *Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction*, 1996.
21: Björn Gambäck, *Processing Swedish Sentences: A Unification-Based Grammar and Some Applications*. June 1997.
22: Klas Orsvärn, *Knowledge Modelling with Libraries of Task Decomposition Methods*, 1996.
23: Kia Höök, *A Glass Box Approach to Adaptive Hypermedia*, 1996.
24: Bengt Ahlgren, *Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption*, 1997.
25: Johan Montelius, *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*, May, 1997.
26: Jussi Karlgren, *Stylistic experiments in information retrieval*, 2000.
27: Ashley Saulsbury, *Attacking Latency Bottlenecks in Distributed Shared Memory Systems*, 1999.
28: Kristian Simsarian, *Toward Human Robot Collaboration*, 2000.
29: Lars-Åke Fredlund, *A Framework for Reasoning about Erlang Code*, 2001.
30: Thiemo Voigt, *Architectures for Service Differentiation in Overloaded Internet Servers*, 2002.